Optimal algorithms for selecting top-k combinations of attributes : theory and applications

Lin, Chunbin

2018-02

Lin , C , Lu , J , Wei , Z , Wang , J & Xiao , X 2018 , ' Optimal algorithms for selecting top-k combinations of attributes : theory and applications ' , VLDB Journal , vol. 27 , no. 1 , pp. 27-52 . https://doi.org/10.1007/s00778-017-0485-2

http://hdl.handle.net/10138/233871
https://doi.org/10.1007/s00778-017-0485-2

submittedVersion

Noname manuscript No.
(will be inserted by the editor)

Optimal Algorithms for Selecting Top-k Combinations of Attributes: Theory and Applications

Chunbin Lin · Jiaheng Lu · Zhewei Wei · Jianguo Wang · Xiaokui Xiao

Received: date / Accepted: date

Abstract Traditional top-k algorithms, e.g., TA and NRA, have been successfully applied in many areas such as information retrieval, data mining, and databases. They are designed to discover k objects, e.g., top-k restaurants, with highest overall scores aggregated from different attributes, e.g., price and location. However, new emerging applications like query recommendation, require providing the best combinations of *attributes*, instead of objects. The straightforward extension based on the existing top-k algorithms is prohibitively expensive to answer top-k combinations because they need to enumerate all the possible combinations, which is exponential to the number of attributes.

In this article, we formalize a novel type of top-k query, called top-k, m, which aims to find top-k combinations of attributes based on the overall scores of the top-m objects within each combination, where m is the number of objects forming a combination. We propose a family of efficient top-k, m algorithms with different data access methods, i.e., sorted accesses and random accesses and different query certainties, i.e., exact query processing and approximate query processing. Theoretically, we prove that our algorithms are instance-optimal and analyze the bound of the depth of accesses. We further develop optimizations for efficient query evaluation to reduce the computational and the memory costs and the number of accesses. We provide a case study on the real applications of top-k, m queries for an online biomedical search engine. Finally, we perform comprehensive experiments to demonstrate the scalability and efficiency of top-k, m algorithms on multiple real-life datasets.

This work is partially supported by NSF BIGDATA 1447943, Academy of Finland (310321), NSF China (61472427,61502503), DSAIR center in NTU and Grant MOE2015-T2-2-069 Singapore.

C. Lin
Department of Computer Science and Engineering, University of California, San Diego, USA
E-mail: chunbinlin@cs.ucsd.edu

J. Lu
Department of Computer Science, University of Helsinki, Finland
E-mail: jiaheng.lu@helsinki.fi

Z. Wei
School of Information, Renmin University of China, China
E-mail: zhewei@ruc.edu.cn

J. Wang
Department of Computer Science and Engineering, University of California, San Diego, USA
E-mail: csjianguo@cs.ucsd.edu

X. Xiao
School of Computer Science and Engineering, Nanyang Technological University, Singapore
E-mail: xkxiao@ntu.edu.sg

1 Introduction

Efficient processing of top-k queries is a crucial requirement in many applications involving massive amounts of data. Traditional top-k algorithms [4,5,9,10,18,22,29,39,38] have obtained great success in finding k independent objects with highest overall scores aggregated from different attributes. For instance, given two ranked lists of prices and locations for restaurants, existing top-k algorithms are efficient in finding top-k restaurants with highest overall scores of prices and locations.

However, many applications in recommendation systems require finding k combinations instead of k independent objects with highest overall scores. For example, given a collection of clothes, shoes and watches, each item is associated with a ranked list of (*UserID*, *Score*) pairs[1]. A recommendation task is to recommend the best (*cloth*, *shoe*, *watch*) combination to maximize the overall scores of users who purchased this combination before. Note that, the scores from users who have purchased the whole combination (not only a

[1] Lists are sorted by scores decreasingly.

single item) are important, as they consider not only the individual factors of each item, e.g., price, but also the holistic factors like visual appearance [16], e.g., best matched colors and styles. In this paper, we model such combination selection as *top-k,m* problems, which find top-k combinations with the highest overall scores based on the scores of their top-m objects (e.g., top-m users) by a monotonic aggregate function (e.g., sum).

Let us consider another example of NBA data in Figure 1: there are three groups, i.e., *forward*, *center*, and *guard*, and each group contains multiple athletes. Each athlete is associated with a list of (gameID, score) pairs[2]. For example, $(G01, 9.31)$ in the F_1 list means the athlete attended game $G01$ and got an overall score 9.31. Assume a basketball coach plans to select a good (*forward*,*center*,*guard*) combination to build a mini-team for a competition by considering their historical performance in games. Suppose the coach considers the sum of the scores of top-2 games for each possible combination. We can model this problem as a top-k, m problem again, i.e., it selects the top-k combinations of athletes according to their best top-m aggregate scores for games *where they played together*. In this example, it is a top-$1, 2$ problem. As illustrated in Figure 1, $F_2C_1G_1$ is the best combination of athletes since the top-2 games in which the three athletes played together are $G02$ and $G05$, and 40.27 $(= 21.51 + 18.76)$ is the highest overall score (w.r.t. the sum of the top-2 scores) among all eight combinations.[3] Therefore, we say that the answer of the top-$1, 2$ query in Figure 1 is the combination "$F_2C_1G_1$".

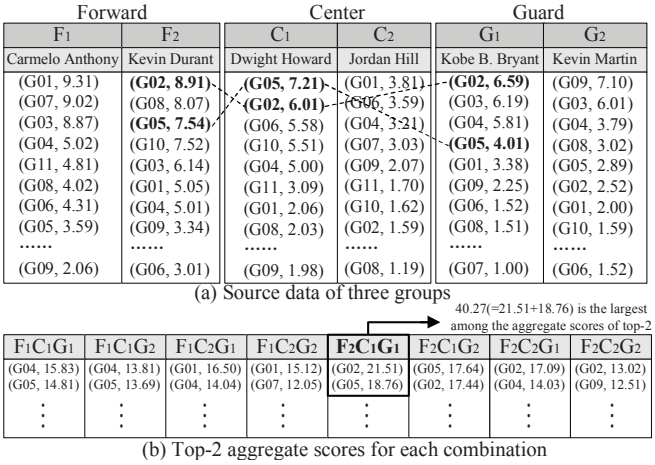
Top-k, m problems have many other real-life applications such as *trip selection* and *keyword query refinement*, which will be described in details in Section 4.

Challenges. To answer a top-k, m query, one method (a baseline approach) is to extend the state-of-the-art top-k algorithms, e.g. the threshold algorithm (TA) [10] in the following way: (Step 1) Enumerate all the possible combinations. (Step 2) Obtain the top-m objects and their associated scores for each combination based on TA. (Step 3) Calculate the scores by aggregating the scores of the top-m objects for each combination and return the top-k combinations with highest overall scores. The main limitation of the baseline method is that it needs to compute the top-m objects for each combination. The final results cannot be returned unless all the top-m objects are obtained for each combination. In this paper, we propose a new family of efficient top-k, m algorithms which avoid the expensive computation of top-m objects of each combination.

Contributions. The key contributions of this article are as follows :

[2] The score is computed by an aggregation of various scoring items provided by the NBA for the corresponding game.

[3] The top-2 games of each combination is shown in Figure 1(b).



(a) Source data of three groups

(b) Top-2 aggregate scores for each combination

Fig. 1 Example NBA data. The answer for the top-$1, 2$ query is $F_2C_1G_1$. Values in bold font indicate tuples contributing to the score of this best combination.

1. We propose a new type of top-k query, called top-k,m query, targeting at finding best k attribute combinations according to the overall scores of the corresponding top-m objects. To demonstrate the applicability of top-k,m queries, we describe several real-life applications.
2. We study the top-k,m queries in scenarios where both sorted accesses and random accesses are allowed. We show that the baseline method ETA, which extends the state-of-the-art top-k algorithm TA (threshold algorithm), is not instance optimal. Then we propose two provably instance optimal algorithms ULA and ULA$^+$, where ULA avoids the need of computing top-m objects for each combination by judiciously calculating the upper bound and lower bound for them, while ULA$^+$ adds a series of optimization methods into ULA to prune away useless combinations without reading any tuples in the associated lists and avoid useless sorted and random accesses in lists. In addition, we show that the optimality ratios of ULA and ULA$^+$ are tight. Furthermore, we provide a deep analysis of the expected depth of accesses for ULA and ULA$^+$, which can be viewed as a quantitative analysis result to the instance optimality.
3. We investigate top-k,m queries where only sorted accesses are allowed, i.e., random accessed are forbidden. We show that the baseline method ENRA, which extends the state-of-the-art top-k algorithm NRA, is not instance optimal. Therefore, we propose two provably instance optimal algorithms NULA and NULA$^+$ where NULA$^+$ applies new optimizations to NULA in order to avoid accessing unnecessary lists and computing unnecessary bounds. Besides the instance optimality, we also prove that the optimality ratios of NULA and NULA$^+$ are tight.
4. We extend our top-k,m algorithms (ULA, ULA$^+$, NULA and NULA$^+$) and the baseline methods (ETA and ENRA) to the approximate environment where the exact top-k,m

answers are not required. We prove that the approximate algorithms extending from ULA, ULA$^+$, NULA, NULA$^+$ are instance optimal.
5. We provide a case study on biomedical query refinement to demonstrate how to apply top-k,m algorithms into real-life problems.
6. Finally, we verify the efficiency and scalability of our algorithms using four real-life data sets, including NBA data, YQL trip-selection data, XML data and biomedical data. We find that our top-k,m algorithms result in order-of-magnitude performance improvements when compared to baseline algorithms.

Paper Organization. The rest of this article organized as follows. We describe the related works in Section 2. Section 3 formally defines the top-k,m problem and Section 4 lists real-life applications of top-k,m problems. We propose top-k,m algorithms supporting sorted access and random access in Section 5. In Section 6, we introduce the top-k,m problems with no random accesses and propose efficient algorithms. We study the approximate top-k,m problem in Section 7. Section 8 describes an application scenario in details. We conduct experiments to evaluate the performance of all the algorithms in Section 9. Finally, we conclude this article in Section 10.

2 Related work

In this section, we review the related works on top-k algorithm with multiple access models, and then describe the new contributions of this journal article compared to our previous conference version [26] .

2.1 Existing top-k algorithms

Top-k queries were studied extensively in many areas including relational databases, XML data and graph data [3,4,7, 8,10–12,17,20,23,24,28,31–34,37,40–43,45,46]. Notably, Fagin et al. [10] present a comprehensive study of various methods for top-k aggregation of ranked inputs. They identify two types of accesses to the ranked lists: *sorted accesses* and *random accesses*. In particular, sorted accesses read the tuple of lists sequentially and random accesses quickly locate tuples whose ID has been seen by sorted access[4]. For example, in Figure 1, at depth 1 (depth d means the number of tuples seen under sorted access to a list is d), consider the combination "$F_2C_1G_1$"; the tuples seen by sorted access are $(G02, 8.91)$, $(G05, 7.21)$, $(G02, 6.59)$ and we can quickly locate all tuples (i.e., $(G02, 6.01)$, $(G05, 7.54)$, $(G05, 4.01)$) whose IDs are $G02$ or $G05$ by random accesses.

[4] Hash indexes can be built to achieve the goal of random accesses.

2.1.1 Top-k algorithms with sorted and random accesses

For the case where both sorted and random accesses are possible, a threshold algorithm (TA) [10] (independently proposed in [31, 14]) retrieves objects from the ranked inputs in a round-robin fashion and directly computes their aggregate scores by using random accesses to the lists where the object has not been seen. Fagin et al. prove that TA is an instance-optimal algorithm.

In this article, we study the top-k,m problem where both sorted accesses and random accesses are allowed (Section 5). Note that the straightforward extension of TA is inefficient because it needs to enumerate all the possible combinations.

2.1.2 Top-k algorithms with no random accesses

There is a rich literature for top-k queries in scenarios where random accesses are not allowed (e.g. [13,30,10]). The first algorithm that only allows sorted access is Stream-combine (SC) proposed in [13]. SC reports only objects which have been seen in all sources. In addition, an object is reported as soon as it is guaranteed to be in the top-k set. In other words, the algorithm does not wait until the whole top-k result has been computed in order to output it, but provides the top-k objects with their scores on-line.

[10] proposes an algorithm called "no-random accesses" (NRA), which presents stronger stop condition. NRA iteratively retrieves objects from the ranked inputs in a round-robin fashion, and maintains the upper and lower bounds for those objects, the final results are guaranteed if the lower bounds of the objects in W_k (a set of k objects with highest lower bounds) are larger than the upper bounds of the other objects outside W_k. A difference between SC and NRA is that SC does not maintain W_k, but only the top-k objects with the highest upper bounds.

[30] proposes a more generic rank aggregation operator J*, which is appropriate for merging ranked inputs based on a join condition on attributes other than the scores. J* can be used as an operator in a query plan which joins multiple ranked inputs. However, [18] shows that J* is less efficient than NRA for top-k queries and provides a "partially" non-blocking version of NRA, called NRA-RJ, which outputs an object as soon as it is guaranteed to be in the top-k (like SC), however, without necessarily having computed its exact aggregate score (like NRA). If exact aggregate scores are required, [19] proposes another version of NRA that outputs exact scores on-line (like SC) and can be applied for any join predicate (like J*). This algorithm uses a threshold which is inexpensive to compute, appropriate for generic rank join predicates. However, it incurs more object accesses than necessary in top-k queries.

Another example of no random access top-k algorithms is LARA proposed by Mamoulis et al. [27], which imposes

two phases (*growing* and *shrinking*) that any top-k algorithm with *no random accesses* (including NRA, SC, J*, NRA-RJ) should go through. In the *growing phase*, the set of top-k candidates grows and no pruning can be performed. However, in the *shrinking phase*, new accessed objects would not be stored anymore, and the set of candidates shrinks until the top-k result is finalized. The condition to transform from *growing phase* to *shrinking phase* is that the smallest lower bound in W_k is no less than the current threshold value. In addition, LARA employs a lattice-based data structure to keep a leader object for each subset of the ranked inputs, and leader objects provide upper bound scores for objects that have not been seen yet on their corresponding inputs.

In this article, we study the problem of top-k,m queries with *no* random accesses (Section 6), which cannot be efficiently answered by the existing top-k algorithms, say NRA, SC, J* and LARA.

2.1.3 Other top-k algorithms

There is also a rich literature for top-k queries in other environments, such as (1) no sorted access on restricted lists [4, 5], (2) ad-hoc top-k queries [22,44] and (3) no need for exact aggregate score [18,29,39]. For more information about top-k query evaluation, readers may refer to an excellent survey paper [21]. In this article, we study the approximate top-k,m problems where only approximate answers are needed (Section 7). We propose instance optimal algorithms that produce approximate answers with error guarantees.

2.2 Compared with the previous preliminary version

This article is an extension from our previous conference version [26]. This work substantially improves the previous version by adding amount of non-trivial new contributions, including new top-k,m problems and algorithms (Section 6, 7 and 8), theoretical results (Section 5.5), and new experiments (Section 9).

3 Problem Formulation

Given a set of groups G_1,...,G_n where each group G_i contains multiple elements e_{i1},...,e_{il_i}, we assume that each element e is associated with a ranked list L_e, where each tuple $\tau \in L_e$ is composed of an ID $\rho(\tau)$ and a score $\sigma(\tau)$. The list is ranked by the scores in descending order. Let $\epsilon = (e_{1i}, e_{nj}) \in G_1 \times \cdots \times G_n$ denote an element of the cross-product of the n groups, hereafter called *combination*. For instance, recall Figure 1, every three athletes from different groups form a combination (e.g., {Kevin Durant, Dwight Howard, Kobe B. Bryant}).

Given a combination ϵ, a *match instance* $\mathcal{I}^\epsilon$ is defined as a set of tuples based on some arbitrary join condition on IDs of tuples from lists. Each tuple in a match instance should come from different groups. As seen in Figure 1, given a combination {Kevin Durant, Dwight Howard, Kobe B. Bryant}, $\{(G01, 9.31), (G01, 3.81), (G01, 3.38)\}$ is a match instance for the game $G01$. Further, we define two aggregate scores: *tScore* and *cScore*, that is, the score of each match instance $\mathcal{I}^\epsilon$ is calculated by *tScore*, and the top-m match instances are aggregated to obtain the overall score, called *cScore*. More precisely, given a match instance $\mathcal{I}^\epsilon$ defined on ϵ,

$$tScore(\mathcal{I}^\epsilon) = \mathcal{F}_1(\sigma(\tau_1), \ldots, \sigma(\tau_n))$$

where $\mathcal{F}_1$ is a function: $\mathbb{R}^n \to \mathbb{R}$ and $\tau_1, \ldots, \tau_n$ form the matching instance $\mathcal{I}^\epsilon$. Further, given an integer m and a combination ϵ,

$$cScore(\epsilon, m) = \max\{\mathcal{F}_2(tScore(\mathcal{I}_1^\epsilon), \ldots, tScore(\mathcal{I}_m^\epsilon))\}$$

where $\mathcal{F}_2$ is a function $\mathbb{R}^m \to \mathbb{R}$ and $\mathcal{I}_1^\epsilon, \ldots, \mathcal{I}_m^\epsilon$ are any m distinct match instances defined on the combination ϵ. Intuitively, $cScore$ returns the maximum aggregate scores of m match instances. Following common practice (e.g., [10]), we require both $\mathcal{F}_1$ and $\mathcal{F}_2$ functions to be monotonic, i.e., the greater the individual score, the greater the aggregate score. This assumption captures most practical scenarios, e.g., if one athlete has a higher score (and the other scores remain the same), then the whole team is better.

Definition 1 (top-k,m problem) Given groups $G_1, \ldots, G_n$, two integers k, m, and two score functions $\mathcal{F}_1$, $\mathcal{F}_2$, the *top-k,m problem* is an (n+4)-tuple $(G_1, \ldots, G_n, k, m, \mathcal{F}_1, \mathcal{F}_2)$. A solution is an ordered set $\mathcal{S}$ containing the top-k combinations $\epsilon = (e_{1i}, \ldots, e_{nj}) \in G_1 \times \cdots \times G_n$ ordered by $cScore(\epsilon, m)$.

Example 1 Consider a top-1, 2 query on Figure 1, and assume that $\mathcal{F}_1$ and $\mathcal{F}_2$ are *sum*. The final answer $\mathcal{S}$ is $\{F_2C_1G_1\}$. This is because the top-1 match instance $\mathcal{I}_1$ of $F_2C_1G_1$ consists of tuples $(G02, 8.91)$, $(G02, 6.01)$ and $(G02, 6.59)$ of the game $G02$ with $tScore$ $21.51 = 8.91 + 6.01 + 6.59$. And the top second instance $\mathcal{I}_2$ consists of tuples whose game ID is $G05$ with $tScore$ $18.76 = 7.54 + 7.21 + 4.01$. Therefore, the $cScore$ of $F_2C_1G_1$ is $40.27 = 21.51 + 18.76$, which is the highest score among that of all combinations. □

Novelty of top-k, m. It is important to note that top-k, m problems *cannot* be reduced to existing top-k problems. The essential difference between traditional top-k queries and our top-k,m problem is that the top-k, m problem returns the top-k *combinations* of elements, but the top-k problem returns the top-k *objects*. Therefore, a top-k,m problem *cannot* be converted to a top-k problem through a careful choice of the

aggregate function. In addition, contrarily to the top-k problem, a top-k,m query also *cannot* be transformed into a SQL (nested) query, since SQL queries return tuples but our goal is to return *element combinations* based on ranked inverted lists, which is not supported in SQL language. Therefore, our top-k,m work, which focuses on selecting and ranking sets of elements, is a highly non-trivial extension of the traditional top-k problem.

To have a better understanding of top-k,m queries, we can treat a top-k,m query as two different top-k join queries executed in a pipelined way. More precisely, the first join operation is to find m objects by joining tables with IDs and tables with (ID, scores). Then the second join operates self-join on the result of the first join in order to get scores for all the combinations, then the top-k results are added to the result set. Note that the straightforward implementation of these two joins to answer a top-k,m query will result in the problem of efficiency. In Section 5, we will propose several optimized algorithms to solve this problem.

4 Applications

In this section, we provide several real application scenarios of top-k,m queries to shed some light on the generality and importance of top-k,m models in practice.

Application 1. Top-k,m queries have applications in recommendation systems, e.g., the *trip recommendation* [25,36] and the *dress collocation recommendation* [15,16].

- **Trip recommendation.** Consider a tourist who is interested in planing a trip by choosing one hotel, one shopping mall, and one restaurant in a city. Assume that we have survey data provided by users who made trips before. The data include three groups and each group have multiple attributes (i.e., names of hotels, malls, or restaurants), each of which is associated with a list of users' IDs and grades. Top-k,m queries recommend top-k trips which are combinations of hotels, malls, and restaurants based on the aggregate value of the highest m scores of the users who had the experience of this exact trip combination.
- **Dress collocation recommendation.** Consider a customer who wants to find a best (cloth, shoe, watch) combination. Suppose we have the purchased historical data in Figure 2. The recommendation task is to recommend the best (cloth, shoe, watch) combination based on the overall scores of the most significant users who purchased this combination before. Top-k,m queries recommend top-k combinations based on the aggregate value of the highest m scores of the users who had purchased this combination. For example, the best combination in Figure 2 for a top-1,2 query is $C_1S_2W_1$.

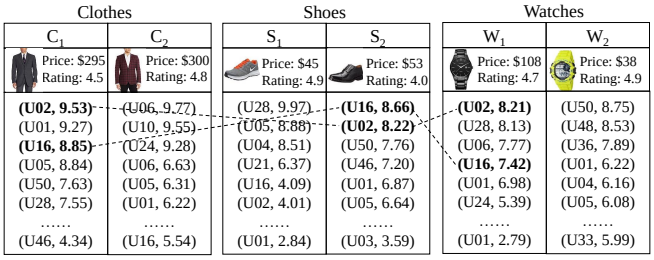
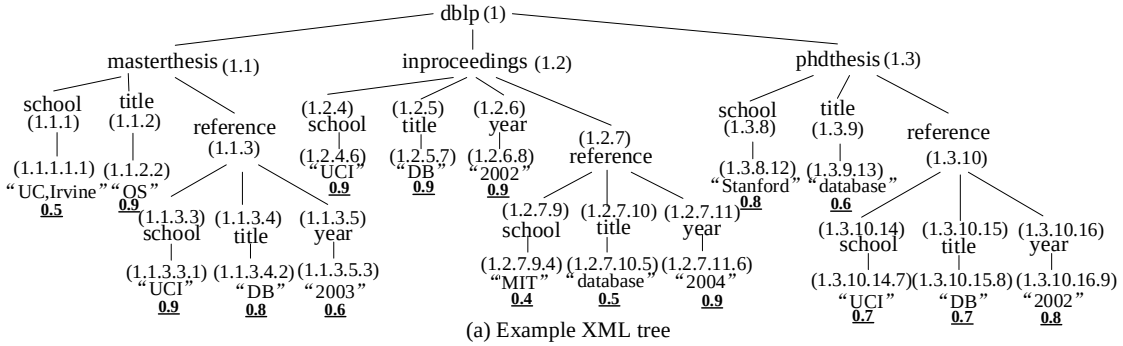


Fig. 2 Motivating example using Amazon data. Our purpose is to choose one item from each of the three groups. The best combination is $C_1S_2W_1$, which achieves the highest overall scores by considering their visual appearances, prices and ratings.

Application 2. Top-k,m queries are also useful in *keyword query rewriting* for search engines and databases. During the last decade, there is an emerging trend of using keyword search in relational and XML databases for better accessibility to novice users. But in a real application, it is often the case that a user issues a keyword query Q which does not return the desired answers due to the mismatch between terms in the query and in documents. A common strategy for remedying this is to perform some query rewriting, replacing query terms with synonyms that provide better matches. Interestingly, top-k,m queries find an application in this scenario. Specifically, for each keyword (or phrase) q in Q, we generate a group $G(q)$ that contains the alternative terms of q according to a dictionary which contains *synonyms* and *abbreviations* of q. For example, see Figure 3 for an example data tree in an XML database. Given a query $Q = \langle \mathsf{DB}, \mathsf{UC\ Irvine}, \mathsf{2002} \rangle$, we can generate three groups: $G_1 = \{\text{“DB”}, \text{“database”}\}$, $G_2 = \{\text{“UCI”}, \text{“UC Irvine”}\}$, and $G_3 = \{\text{“2002”}\}$. We assume that each term in $G(q)$ is associated with a list of document IDs or node identifiers (e.g., JDewey IDs [6] in XML databases) and scores (e.g., information-retrieval scores such as tf-idf). The goal of top-k,m queries is to find the top-k combinations (of terms) by considering the corresponding top-m search results in the database. Therefore, a salient feature of the top-k,m model for the refinement of keyword queries is that it guarantees that the suggested alternative queries have high quality results in the database within the top-m answers.

Remark. Generally speaking, top-k,m queries are of use in any context where one is interested in obtaining combinations of attributes associated with ranked lists. Note that the model of top-k,m queries offers great flexibility in problem definitions to meet the various requirements that applications may have, in particular in the adjustment of the m parameter. For example, in the application to XML keyword search, a user is often interested in browsing only the top few results, say 10, which means we can let $m = 10$ to guarantee the search quality of the refined keywords. In another application, e.g., trip recommendation, if a tourist wants to consider the average score of all users, then we can define m to be large enough



(a) Example XML tree

Group1

DB	database
(1.2.5.7, 0.9) (1.1.3.4.2, 0.8) (1.3.10.15.8, 0.7)	(1.3.9.13, 0.6) (1.2.7.10.5, 0.5)

Group2

UCI	UC, Irvine
(1.1.3.3.1, 0.9) (1.2.4.6, 0.9) (1.3.10.14.7, 0.7)	(1.1.1.1, 0.5)

Group3

2002
(1.2.6.8, 0.9) (1.3.10.16.9, 0.8)

(b) Sorted lists and groups

Fig. 3 An example illustrating XML query refinement using the top-k,m framework. The original query $Q = \langle \mathsf{DB}, \mathsf{UC\ Irvine}, 2002 \rangle$ is refined into $\langle \mathsf{DB}, \mathsf{UCI}, 2002 \rangle$. Each term is associated with an inverted list with the IDs and weights of elements. Underlined numbers in the XML tree denote term scores.

to take the scores of all users into accounts. (Of course, in this case, the number of accesses and the computational cost are higher.)

5 Top-k,m algorithms with sorted and random accesses

In this section we study the top-k,m problems in the scenarios where both sorted accesses and random accesses are allowed.

5.1 The baseline algorithm: ETA

To answer a top-k,m query, one straightforward method (called extended TA, or ETA for short) is to first compute all top-m results for each combination by some well-known algorithms like the threshold algorithm TA [10] and then pick the top-k combinations. However, this method has one obvious shortcoming: it needs to compute top-m results for *each* combination and reads *more* inputs than needed. For example, in Figure 1, ETA needs to compute the top-2 scores for all eight combinations (see Figure 1(b)). Indeed, this method is *not* an *instance-optimal* solution in this context. To address this problem, we develop a set of *provably optimal* algorithms to efficiently answer top-k,m queries.

5.2 Top-k,m algorithm: ULA

When designing an efficient top-k,m algorithm, informally, we observe that a combination ϵ cannot contribute to the final answer if *there exist k distinct combinations whose lower bounds are greater than the upper bounds of ϵ*. To understand this, consider the top-1,2 query in Figure 1 again. At depth 1, for the combination "$F_2C_1G_1$", we get two match instances $G02$ and $G05$ through sorted and random accesses. Then the lower bound of the aggregate score (i.e., $cScore$) of "$F_2C_1G_1$" is at least 40.27 (i.e., $(7.54 + 7.21 + 4.01) + (8.91 + 6.01 + 6.59)$). At this point, we can claim that some combinations are not part of answers. This is the case of "$F_2C_2G_1$", whose $cScore$ is no more than 38.62 ($= 2 \times (8.91 + 3.81 + 6.59)$). Since $38.62 < 40.27$, $F_2C_2G_1$ cannot be the top-1 combination. We next formalize this observation by carefully defining lower and upper bounds of combinations. We start by presenting threshold values, which will be used to estimate the upper bounds for the unseen match instances.

Definition 2 (threshold value) Let $\epsilon = (e_{1i}, \ldots, e_{nj}) \in G_1 \times \cdots \times G_n$ be an arbitrary combination, and τ_i the current tuple seen under sorted access in list L_i. We define the *threshold value* $\mathcal{T}^\epsilon$ of the combination ϵ to be $\mathcal{F}_1(\sigma(\tau_1), \ldots, \sigma(\tau_n))$, which is the upper bound of $tScore$ for any unseen match instance of ϵ.

As an example, in Figure 1(a), consider the combination ϵ = "$F_2C_1G_1$", at depth 1. The current tuples are $(G02, 8.91)$, $(G05, 7.21)$, $(G02, 6.59)$. Assume $\mathcal{F}_1 = sum$, we have for threshold value $\mathcal{T}^\epsilon = 8.91 + 7.21 + 6.59 = 22.71$.

Definition 3 (lower bound) Assume one combination ϵ has seen m' distinct match instances. Then the *lower bound* of the $cScore$ of ϵ is computed as follows:

$$\epsilon^{\min} = \begin{cases} \mathcal{F}_2(tScore(\mathcal{I}_1^\epsilon), \ldots, tScore(\mathcal{I}_{m'}^\epsilon), \underbrace{0, \ldots, 0}_{m-m'}) & m' < m \\ \max\{\mathcal{F}_2(\underbrace{tScore(\mathcal{I}_i^\epsilon), \ldots, tScore(\mathcal{I}_j^\epsilon)}_{m})\} & m' \geq m \end{cases}$$

When $m' < m$, we use the minimal score (i.e., zero) of unseen $m - m'$ match instances to estimate the lower bound of the $cScore$. On the other hand, when $m' \geq m$, $\epsilon^{\min}$ equals the maximal aggregate scores of m match instances.

Definition 4 (upper bound) Assume one combination ϵ has seen m' distinct match instances, where there are m'' match instances ($m'' \leq m'$) whose scores are *greater than or equal to* $\mathcal{T}^{\epsilon}$. Then the *upper bound* of the $cScore$ of ϵ is computed as follows:

$$\epsilon^{\max} = \begin{cases} \mathcal{F}_2(tScore(\mathcal{I}^{\epsilon}_1), \ldots, tScore(\mathcal{I}^{\epsilon}_{m''}), \underbrace{\mathcal{T}^{\epsilon}, \ldots, \mathcal{T}^{\epsilon}}_{m-m''}) & m'' < m \\ \max\{\mathcal{F}_2(\underbrace{tScore(\mathcal{I}^{\epsilon}_i), \ldots, tScore(\mathcal{I}^{\epsilon}_j)}_{m})\} & m'' \geq m \end{cases}$$

If $m'' < m$, it means that there is still a chance that we will see a new match instance whose $tScore$ contributes to the final $cScore$. Therefore, the computation of $\epsilon^{\max}$ should be padded with $m - m''$ copies of the threshold value (i.e., $\mathcal{T}^{\epsilon}$), which is the upper bound of $tScore$ for all unseen match instances. Otherwise, $m'' \geq m$, meaning that the final top-m results are already seen and thus $\epsilon^{\max} = cScore(\epsilon, m)$ now.

Example 2 This example illustrates the computation of the upper and lower bounds. See Figure 1 again. Assume $\mathcal{F}_1$ and $\mathcal{F}_2$ are sum, and the query is top-$1, 2$. At depth 1, the combination "$F_2C_1G_1$" read tuples $(G02, 8.91)$, $(G05, 7.21)$, and $(G02, 6.59)$ by sorted accesses, and $(G05, 7.54)$, $(G02, 6.01)$, $(G05, 4.01)$ by random accesses. $m' = m = 2$. Therefore, the current lower bound of "$F_2C_1G_1$" is 40.27 (i.e., $(7.54+7.21+4.01)+(8.91+6.01+6.59) = 18.76+21.51$), since the two match instances of $F_2C_1G_1$ are $G02$ and $G05$. The threshold $\mathcal{T}^{F_2C_1G_1} = 8.91 + 7.21 + 6.59 = 22.71$ and $m'' = 0$, since $18.76 < 22.71$ and $21.51 < 22.71$. Therefore, the upper bound is 45.42 (i.e., $22.71 + 22.71$). In fact, the final $cScore$ of "$F_2C_1G_1$" is exactly 40.27 which equals the current lower bound. Note that the values of lower and upper bounds are dependent of the depth where we are accessing. For example, at depth 2, the upper bound of "$F_2C_1G_1$" decreases to 41.78 (i.e., $21.51 + 20.27$) and the lower bound remains the same. □

The following lemmas show how to use the above bounds to determine if a combination ϵ can be pruned safely or confirmed to be an answer.

Lemma 1 (drop-condition) *One combination ϵ does not contribute to the final answers if there are k distinct combinations $\epsilon_1, \ldots, \epsilon_k$ such that $\epsilon^{\max} < \min\{\epsilon^{\min}_i \mid 1 \leq i \leq k\}$.*

Proof The aggregate score of the top-m match instances is no more then the upper bound of ϵ, i.e., $cScore(\epsilon, m) \leq \epsilon^{\max}$. And $\forall i \in [1, k]$, $cScore(\epsilon_i, m) \geq \epsilon^{\min}_i$ holds, since the $\epsilon^{\min}_i$ is the lower bound of ϵ_i. Therefore, $cScore(\epsilon, m) < \min\{cScore(\epsilon'_i, m) \mid 1 \leq i \leq k\}$, which means that ϵ cannot be one of the top-k answers, as desired. ■

Lemma 2 (hit-condition) *One combination ϵ should be an answer if there are at least $N_{\text{com}} - k$ (N_{com} is the total number of combinations) distinct combinations $\epsilon_1, \ldots, \epsilon_{N_{\text{com}}-k}$, such that $\epsilon^{\min} \geq \max\{\epsilon^{\max}_i \mid 1 \leq i \leq N_{\text{com}} - k\}$.*

Proof The aggregate score of the top-m match instances of ϵ is no less than the lower bound of ϵ, i.e., $cScore(\epsilon, m) \geq \epsilon^{\min}$. And $\forall i \in [1, N_{\text{com}} - k]$, $\epsilon^{\max}_i \geq cScore(\epsilon_i, m)$. Therefore, $cScore(\epsilon, m) \geq \max\{cScore(\epsilon_i, m) \mid 1 \leq i \leq N_{\text{com}} - k\}$, meaning that the top-$m$ aggregate score of ϵ is larger than or equal to that of other $N_{\text{com}} - k$ combinations. Therefore ϵ must be one of the top-k,m answers. ■

Definition 5 (termination) A combination ϵ can be *terminated* if ϵ meets one of the following conditions: (i) the drop-condition, (ii) the hit-condition, or (iii) ϵ has seen m match instances whose $tScore$s are greater than or equal to the threshold value $\mathcal{T}^{\epsilon}$.

Intuitively, one combination is terminated if we do not need to compute its lower or upper bounds any further. The first two conditions in the above definition are easy to understand. The third condition means that we have found top-m match instances of ϵ. Note that we may not see top-m match instances even if ϵ satisfy the drop- or hit-condition.

We are now ready to present a novel algorithm named **ULA** (**U**pper bound and **L**ower bound **A**lgorithm), that relies on the upper and lower bounds of combinations. The ULA algorithm is shown in Algorithm 1.

Algorithm 1: The ULA algorithm

Consider a top-k,m problem instance with n groups $G_1, \ldots, G_n$, where each group has multiple lists $L_{ij} \in G_i$.
(i) Do sorted access in parallel to each of the sorted lists L_{ij}. As a tuple τ is seen under sorted access in some list, do random access to all other lists in G_j ($j \neq i$) to find all tuples τ' such that $\rho(\tau) = \rho(\tau')$.
(ii) For each unterminated combination ϵ (by Definition 5), compute $\epsilon^{\min}$ and $\epsilon^{\max}$, and check if ϵ can be terminated now.
(iii) If there are at least k combinations which meet the hit-condition, then the algorithm halts. Otherwise, go to step (i).
(iv) Let Y be a set containing the k combinations (breaking ties arbitrarily) when ULA halts. Output Y.

Example 3 We continue the example of Figure 1 to illustrate the ULA algorithm. First, in step (i) (at depth 1), ULA performs sorted accesses on one row for each list and does

the corresponding random accesses. In step (ii) (at depth 1 again), it computes the lower and upper bounds for each combination, and then three combinations $F_1C_2G_2$, $F_2C_2G_1$ and $F_2C_2G_2$ are safely terminated, since their upper bounds (i.e., $\epsilon^{\max}_{F_1C_2G_1} = 39.42$, $\epsilon^{\max}_{F_2C_2G_1} = 38.62$ and $\epsilon^{\max}_{F_2C_2G_2} = 39.64$) are less than the lower bound of $F_2C_1G_1$ ($\epsilon^{\min}_{F_2C_1G_1} = 40.27$). Next, we go to step (i) again (at depth 2), as there is no combination satisfying the hit-condition in step (iii). Finally, at depth 4, $F_2C_1G_1$ meets the hit-condition and the ULA algorithm halts. To understand the advantage of ULA over ETA, note that ETA cannot stop at depth 4, since $F_2C_2G_1$ does not yet obtain its top-2 match instances. Indeed, ETA stops at depth 5 with 54 accesses, whereas ULA performs only 50 accesses. □

Theorem 1 *If the aggregation functions $\mathcal{F}_1$ and $\mathcal{F}_2$ are monotone, then ULA correctly finds the top-k,m answers.*

Proof Let Y be the results set in Step (iv) of ULA, we claim that the $cScore$ of each combination $\epsilon \in Y$ is larger than that of $\xi \notin Y$. In ULA, for each combination, the score of any unseen match instance is no more than the threshold value, since $\mathcal{F}_1$ is monotone. Thus, the aggregate score of top-m match instances (i.e., $cScore(\epsilon, m)$) must be distributed in $[\epsilon^{min}, \epsilon^{max}]$, since the $\mathcal{F}_2$ is required to be monotone. Combinations would be added into Y only if they meet hit-condition in Step (iii). Therefore, $\epsilon^{min} \geq \xi^{max} (\epsilon \in Y, \xi \notin Y)$. So we have $cScore(\epsilon, m) \geq cScore(\xi, m)$, since $\epsilon^{min} \leq cSocre(\epsilon, m)$ and $cScore(\xi, m) \leq \xi^{max}$. as desired. ■

Theorem 2 *ULA requires only bounded buffers, whose size is independent of the size of the database.*

Proof Other than a little bit of bookkeeping, all that ULA must remember is the upper bound and lower bound for each combination, and (pointers to) the objects seen at each step.

Discussion. Note that in the ULA algorithm the output set Y is unordered by $cScore$. This is because we do not compute the exact $cScore$ of combinations in the algorithm (which is in fact one advantage of our algorithm). In the case where the output set should be ordered by $cScore$, we need to extend ULA in two aspects. First, in step (ii), if a combination meets hit-condition, then we need to continuously maintain its lower and upper bounds. Second, we add a new step to sort the combinations in Y. We continue to access nodes for combinations in Y and maintain their upper and lower bounds. We progressively output one combination $\epsilon \in Y$ to be the exact top-k' ($k' \leq k$) if ϵ^{min} is no less than $k - k'$ upper bounds of the other combinations in Y. In this way, all combinations can be output in order by $cScore$ values. A merit of this approach is that it still avoids the computation of the exact $cScore$ of combinations (as we will show later, it is still an *instance optimal* algorithm in the class of algorithms with ordered output).

5.3 Optimized top-k,m algorithm: ULA+

In this subsection, we present several optimizations to minimize the number of accesses, memory cost, and computational cost of the ULA algorithm by proposing an extension, called ULA$^+$. In a nutshell, we (i) completely avoid the need of computing bounds for some combinations which are not be part of final answers; and (ii) reduce the number of random accesses and sorted accesses in three different levels.

Pruning combinations without computing the bounds The ULA algorithm has to compute the lower and upper bounds for each combination, which may be an expensive operation when the number of combinations is large. We next propose an approach which prunes away many useless combinations safely *without computing their upper or lower bounds*.

We sort all lists in the same group by the scores of their top tuples. Notice that all lists are sorted by decreasing order. Intuitively, the combinations with lists containing small top tuples are guaranteed not to be part of answers, as their scores are too small. Therefore, we do not need to take time to compute their accurate upper and lower bounds. We exploit this intuitive observation by defining the precise condition under which a combination can be safely pruned without computing its bounds. We first define a relationship between two combinations called *dominating*.

Given a group G in a top-k,m problem instance, let L_e and L_t be two lists associated with attributes $e, t \in G$, we say L_e dominates L_t, denoted $L_e \succ L_t$ if $L_e.\sigma(\tau_m) \geq L_t.\sigma(\tau_1)$, where τ_i denote the ith tuple in the list. That is, the score of the mth tuple in L_e is greater than or equal to the score of the *first* tuple in L_t.

Definition 6 (Dominating) A combination $\epsilon = \{e_1, \ldots, e_n\}$ is said to *dominate* another combination $\xi = \{t_1, \ldots, t_n\}$ (denoted $\epsilon \succeq \xi$) if for every $1 \geq k \geq n$, either $e_i = t_i$ or $L_{e_i} \succeq L_{t_i}$ holds, where e_i and t_i are two (possibly identical) attributes of the same group G_i.

For example, in Figure 4, there are two groups G_1 and G_2. We say that the combination "A_2B_1" dominates "A_3B_2", because in the group G_1, $7.1 > 6.3$ and in G_2, $8.2 > 8.0$. In fact, "A_2B_1" dominates all combinations of attributes from A_3 to A_n in G_1 and from B_2 to B_n in G_2. Note that the lists in each group here are sorted by the scores of the top tuples.

Lemma 3 *Given two combinations ϵ and ξ, if ϵ dominates ξ then the upper bound of ϵ is greater than or equal to that of ξ.*

Proof If ϵ dominates ξ, then for every attribute e in ξ, if $e \notin \epsilon$, then there is an attribute t in ϵ, s.t. the m-th tuple in the list L_e has a larger score than the first tuple in L_t. Therefore, the upper bound of m match instances of ϵ is greater than or equal to that of ξ. More formally, $\epsilon \succeq \xi \Rightarrow \forall i, L_{e_i}.\sigma(\tau_m) \geq$

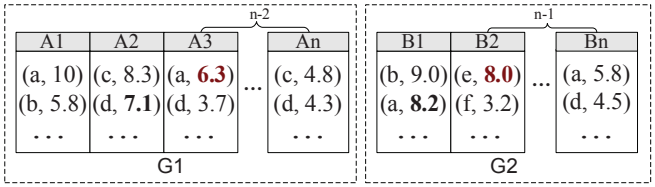


Fig. 4 An example for Lemma 3

$L_{t_i}.\sigma(\tau_1) \Rightarrow \mathcal{F}_1(L_{e_1}.\sigma(\tau_m),\ldots,L_{e_n}.\sigma(\tau_m)) \geq \mathcal{F}_1(L_{t_1}.\sigma(\tau_1),\ldots,L_{t_n}.\sigma(\tau_1))$, since $\mathcal{F}_1$ is monotonic. So $m\times(\mathcal{F}_1(L_{e_1}.\sigma(\tau_m),\ldots,L_{e_n}.\sigma(\tau_m))) \geq m \times (\mathcal{F}_1(L_{t_1}.\sigma(\tau_1),\ldots,L_{t_n}.\sigma(\tau_1)))$. Note that $\epsilon^{\max} \geq m \times (\mathcal{F}_1(L_{e_1}.\sigma(\tau_m),\ldots,L_{e_n}.\sigma(\tau_m)))$, since the threshold value and the scores of the unseen match instances of ϵ are no less than $\mathcal{F}_1(L_{e_1}.\sigma(\tau_m),\ldots,L_{e_n}.\sigma(\tau_m))$. In addition, it is easy to verify that $\xi^{\max} \leq m\times(\mathcal{F}_1(L_{t_1}.\sigma(\tau_1),\ldots,L_{t_n}.\sigma(\tau_1)))$. Therefore, $\epsilon^{\max} \geq \xi^{\max}$ holds, as desired. ■

According to Lemma 3, if ϵ meets the drop-condition (Lemma 1), it means the upper bound of ϵ is small, then any combination ξ which is dominated by ϵ (i.e., ξ's upper bound is even smaller) can be pruned safely and quickly.

To apply Lemma 3 in our algorithm, the lists are sorted in descending order by the score of the first tuple in each list, which can be done off-line. We first access m tuples sequentially for each list and perform random accesses to obtain the corresponding match instances. Then we consider two phases. (i) *Seed combination selection*. As the name indicates, seed combinations are used to trigger the deletion of other useless combinations. We pick the lists in descending order, and construct the combinations to compute their upper and lower bounds until we find one combination ϵ which meets the drop-condition, then ϵ is selected as the seed combination; (ii) *Dropping useless combinations*. By Lemma 3, all combinations which are dominated by ϵ are also guaranteed *not* to contribute to final answers. For each group G_i, assuming that the seed combination ϵ contains the list L_{ai} in G_i, then we find all lists L_{bi} such that $L_{ai} \succeq L_{bi}$. This step can be done efficiently as all lists are sorted by their scores of first tuples. Therefore, all the combinations which are constructed from L_{bi} can be dropped safely without computing their upper or lower bounds.

Example 4 See Figure 4. Assume the query is top-$1,2$ and $\mathcal{F}_1 = \mathcal{F}_2 = sum$. The lists are sorted in descending order according to the score of the first tuple. We access the lists in descending order to find the seed combination, which is $\xi = (A_2, B_1)$ ($\xi^{\max} = 2 \times (7.1 + 8.2) = 30.6 < \epsilon^{\min}$, $\epsilon = \{A_1, B_1\}$). In G_1, $\forall i \in [3,n]$ $L_{A_2} \succ L_{A_i}$ (e.g., $L_{A_2} \succeq L_{A_3}$, since $7.1 > 6.3$). Similarly, in G_2, $\forall i \in [2,n]$ $L_{B_1} \succ L_{B_i}$. Therefore all combinations (A_i, B_j) ($\forall i \in [3,n], j \in [2,n]$), as well as (A_2, B_j) and (B_1, A_i) are dominated by ξ and can be pruned quickly. Therefore there are $(n-2)(n-1) + (n-1) + (n-2) = n^2 - n - 1$ combinations pruned without the (explicit) computation of their bounds, which can significantly save memory and computational costs. □

Note that in the ULA$^+$ algorithm (which will be presented later), we perform the two phases above as a preprocessing procedure to filter out many useless combinations.

Reducing the number of accesses. We now propose some further optimizations to reduce the number of accesses at three different levels: (i) avoiding both sorted and random accesses for specific lists; (ii) reducing random accesses across two lists; and (iii) eliminating random accesses for specific tuples.

Lemma 4 *During query processing, given a list L, if all the combinations involving L are terminated, then we do not need to perform sorted accesses or random accesses upon the list L any longer.*

Proof It is easy to see that when all the combinations involving L are terminated, we cannot find any new combinations involving L to become part of final answers. Therefore, the continuous accesses upon L are useless. ■

If all the accesses upon a list L are terminated, then L can be fan out of the memory, which would save memory cost and computational cost and reduce the number of accesses.

Lemma 5 *During query processing, given two lists L_e and L_t associated with two attributes e and t in different groups, if all the combinations involving L_e and L_t are terminated, then we do not need to perform random accesses between L_e and L_t any longer.*

Proof If all the combinations involving L_e and L_t have been terminated, then we cannot find any new combinations including L_e and L_t to become final answers. Therefore, the random accesses between L_e and L_t are useless. ■

If the random access between lists L_e and L_t is proved to be useless, then all following random accesses between the two lists could be avoided, which could reduce the number of accesses.

Lemma 6 *During query processing, given two lists L_e and L_t associated with two attributes e and t in different groups, consider a tuple τ in list L_e. We say that the random access for the tuple τ from L_e to L_t is useless, if there exists a group G ($e \notin G$ and $t \notin G$) such that $\forall s \in G$, either of the two following conditions is satisfied: (i) the list L_s does not contain any tuple τ', $\rho(\tau) = \rho(\tau')$; or (ii) the combination ϵ involving s, e and t is terminated.*

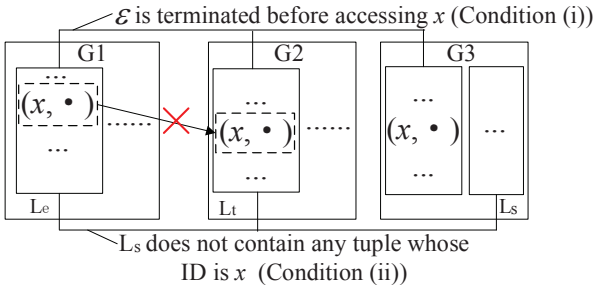


Fig. 5 Example to illustrate Claim 6. Assume there are two lists in group G_3. Random access from L_e to L_t is useless, since ϵ is terminated and L_s does not contain any tuple whose ID is x.

It is not hard to see Lemma 4 and 5 hold. To illustrate Lemma 6, let us consider three groups $G1$, $G2$ and $G3$ in Figure 5, where $G3$ contains only two lists. The list L_s does not contain any tuple whose ID is x and the combination ϵ is terminated. Therefore, according to Lemma 6, the random access between L_e and L_t for tuple x is unnecessary. This is because no match instances of x can contribute to the computation of final answers. Note that it is common in real life that some objects are not contained in some list. For example, think of a player who missed some games in the NBA pre-season. Furthermore, to maximize the elimination of useless random accesses implied in Lemma 6, in our algorithm, we consider the *Small First Access (SFA)* heuristic to control the order of random accesses, that is, we first perform random accesses to the lists in groups with fewer attributes. In this way, the random access across lists in larger groups may be avoided if there is no corresponding tuple in the list of smaller groups. As shown in our experimental results, Lemma 6 and the SFA heuristic have practical benefits to reduce the number of random accesses.

Summarizing, Lemma 4 through 6 imply three levels of granularity to reduce the number of accesses. In particular, Lemma 4 eliminates both random accesses and sorted accesses, Lemma 5 aims at preventing unnecessary random accesses, while Lemma 6 comes in to avoid random accesses for some specific tuples.

In order to exploit the three optimizations in the processing of our algorithm, we carefully design a native data structure named *top-k,m graph* (called KMG hereafter). Figure 6(a) shows an example KMG for the data in Figure 1. Formally, given an instance Π of the top-k,m problem, we can construct a node-labeled, weighted graph $\mathcal{G}$ defined as (V, E, W, C), where (1) V is a set of nodes, each $v \in V$ indicating a list in Π, e.g., in Figure 6, node F_1 refers to the list $F1$ in Figure 1; (2) $E \subseteq V \times V$ is a set of edges, in which the existence of edge (v,v') means that random accesses between v and v' are necessary; (3) for each edge e in E, $W(e)$ is a positive integer, which is the weight of e. The value is the total number of unterminated combinations associated with e; and finally (4) C denotes a collection of subsets of V, each of which indicates a group of lists in Π, e.g., in Figure 6, $C = \{\{F_1, F_2\}, \{C_1, C_2\}, \{G_1, G_2\}\}$. A path of length $|C|$ in $\mathcal{G}$ that spans all subsets of C corresponds to a combination in Π.

Algorithm 2: The ULA$^+$ algorithm

(i) Find the seed combination ϵ and prune all useless combinations dominated by ϵ according to the approach in Section 5.3.
(ii) Initialize a KMG $\mathcal{G}$ for the remaining combinations.
(iii) Do sorted accesses in parallel to each of the lists having nodes in $\mathcal{G}$.
(iv) Do random accesses according to the existing edges in $\mathcal{G}$ (note that we need to first access the smaller group based on SFA strategy). In addition, given a tuple $\tau \in L_n, n \in G$, if there is another group G' such that each node n' in G' (where $\exists$ edge $(n, n') \in \mathcal{G}$) does not contain the tuple with the same ID of τ, then we can immediately stop all random accesses for τ (implied by Lemma 6).
(v) Compute $\epsilon^{\min}$ and $\epsilon^{\max}$ for each unterminated combination ϵ and determine if ϵ is terminated now by Definition 5 using $\epsilon^{\min}$ and $\epsilon^{\max}$. If yes, decrease the weights of all edges involved in ϵ by 1. In addition, remove an edge if its weight is zero and remove a node $v \in \mathcal{G}$ if the degree of v is zero.
(vi) Add ϵ to the result set Y if it meets the hit-condition. If there are at least k combinations which meet the hit-condition, then the algorithm halts. Otherwise, go to step (iii).
(vii) Output the result set Y containing top-k combinations.

Based on the above claims, we propose three dynamic operations in KMG: (i) decreasing the weight of edges by 1 if one of the combinations involving the edge is terminated; (ii) deleting the edge if its weight is 0, which means that random accesses between the two lists are useless (implied by Lemma 5); and (iii) removing the node if its degree is 0, which indicates that both sorted and random accesses in this list are useless (implied by Lemma 4).

Optimized top-k,m algorithm. We are now ready to present the ULA$^+$ algorithm based on KMG, which combines all optimizations implied by Lemma 4 to 6. This algorithm is shown as Algorithm 2.

Example 5 We present an example with the data of Figure 1 to illustrate ULA$^+$. Consider a top-1,2 query again. Firstly, in Step (i), ULA$^+$ performs sorted accesses to two rows of all lists, and finds a seed combination, e.g., $F_2C_1G_2$, as $\epsilon^{\max}_{F_2C_1G_2} = 40.18 < \epsilon^{\min}_{F_2C_1G_1} = 40.27$. Because $L_{C_1} \succ L_{C_2}$, the combination $\epsilon_{F_2C_1G_2}$ dominates $\epsilon_{F_2C_2G_2}$. Therefore, both $\epsilon_{F_2C_1G_2}$ and $\epsilon_{F_2C_2G_2}$ can be pruned in step (i). Then ULA$^+$ constructs a KMG (see Figure 6(a)) for non-pruned combinations in step (ii). Note that there is no edge between F_2 and G_2, since both $\epsilon_{F_2C_2G_2}$ and $\epsilon_{F_2C_1G_2}$ have been pruned. By depth 2, ULA$^+$ computes $\epsilon^{\min}$ and $\epsilon^{\max}$ for each unterminated combination in Step (iii). Then $\epsilon_{F_1C_2G_1}$ and $\epsilon_{F_1C_2G_2}$ meet the drop-condition (e.g., $\epsilon^{\max}_{F_1C_2G_1} = 37.6 < \epsilon^{\min}_{F_2C_1G_1}$), and we decrease the weights by 1 for the corresponding edges, e.g., $w(F_1, G_1) = 1$. In addition, node C_2

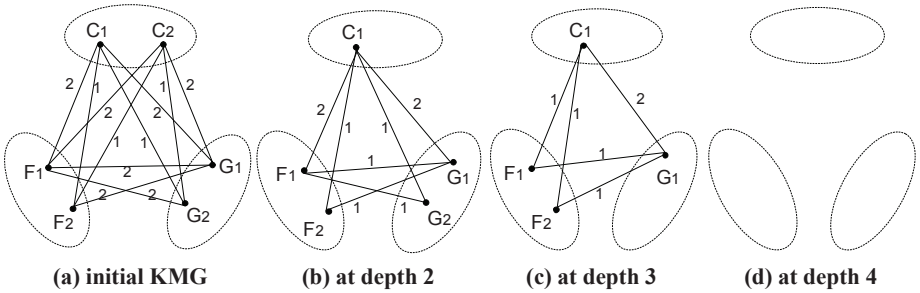


Fig. 6 Example top-k,m graphs (KMG)

should be removed, since all the combinations containing C_2 are terminated, (see Figure 6(b)) in step (iv). At depth 3, $\epsilon_{F_1C_1G_2}$ is terminated, since $\epsilon^{\max}_{F_1C_1G_2} = 36.48 < \epsilon^{\min}_{F_2C_1G_1}$, and we decrease the weights of (F_1, C_1), (F_1, G_2) and (C_1, G_2) by 1 and remove the node G_2 (see Figure 6(c)). Finally, ULA$^+$ halts at depth 4 in step (vi) and $F_2C_1G_1$ is returned as the final result in step (vii). To demonstrate the superiority of ULA$^+$, we compare the numbers of accessed objects for three algorithms: ETA accesses 54 tuples and ULA accesses 50 tuples, while ULA$^+$ accesses only 37 tuples. □

Theorem 3 *If the aggregation functions $\mathcal{F}_1$ and $\mathcal{F}_2$ are monotone, then ULA$^+$ correctly finds the top-k,m answers.*

Proof Let Y be the result set outputted by ULA$^+$, we claim that Y is the same to the result set outputted by ULA. This is because of the correctness of Lemma 3 and Lemma 4, 5 and 6, which prune useless combinations and avoid useless sorted and random accesses. Therefore, by the validity of ULA, ULA$^+$ correctly output the answers, as desired. ■

Theorem 4 *The algorithms ULA and ULA$^+$ halt at the same depth, and ULA$^+$ never accesses more objects than ULA does.*

Proof Assume ULA and ULA$^+$ halt at depth d and d', respectively. Then we would show that $d' = d$. ULA$^+$ access no more objects than ULA, and in particular, ULA covers all the objects that ULA$^+$ accessed. ULA halts by depth d, which means that one object in depth d is the key object to identify the correct top-k combinations, by the correctness of ULA$^+$, ULA$^+$ needs to see the object in depth d, otherwise, ULA† errs. So $d' = d$. ■

5.4 Optimality properties

We next consider the optimality of algorithms. We start by defining the optimality measures, and then analyze the optimality in different cases. Some of the proofs are omitted here due to space limitation; most proofs are non-trivial.

Competing algorithms. Let $\mathbb{D}$ be the class of all databases. We define $\mathbb{A}$ of all deterministic correct top-k,m algorithms running on every database $\mathscr{D}$ in class $\mathbb{D}$. Following the access model in [10], an algorithm $\mathscr{A} \in \mathbb{A}$ can support both sorted accesses and random accesses.

A1	A2
(x,10)	(1, 5)
.....	.....
	(n, 5)
	(n+1, 5)
	(n+2, 1)
	.....
	(2n+1,1)

B1	B2
(x,10)	(2n+1, 5)
.....	.....
	(n+2, 5)
	(n+1, 5)
	(n, 1)
	.....
	(1,1)

Fig. 7 Sub-optimality of ETA.

Cost metrics. We consider the number of tuples seen by sorted access and random access as the dominant computational factor. Let $cost(\mathscr{A}, \mathscr{D})$ be the nonnegative performance cost measured by running algorithm $\mathscr{A}$ over database $\mathscr{D}$, which represents the amount of the tuples accessed.

Instance Optimality. We use the notions of instance optimality. We say that an algorithm $\mathscr{A} \in \mathbb{A}$ is *instance optimal* if for every $\mathscr{A}' \in \mathbb{A}$ and every $\mathscr{D} \in \mathbb{D}$ there exist two constants c and c' such that $cost(\mathscr{A}, \mathscr{D}) \leq c * cost(\mathscr{A}', \mathscr{D}) + c'$.

First, we prove that ETA is not instance optimal in top-k,m problem.

Proof **(Sub-optimality of the ETA algorithm)** We now construct a case to demonstrate that the ETA algorithm is not instance optimal. Assume the query is top-1,1 and the aggregate functions $\mathcal{F}_1$ and $\mathcal{F}_2$ are sum. Consider the database in Figure 7. In order to compute $cScore$ for all the combinations, ETA needs to get the exact top-1 match instance for each combination. So ETA needs to access the tuple $(n+1, 5)$ at depth $n+1$ in lists A_2 and B_2 (see the red boxes) to get the top-1 match instance (i.e., $(n+1, 10)$) for combination $\epsilon_{A_2B_2}$. However, there exists a deterministic algorithm $\mathscr{A}$ halts after accessing the first tuples of each list by depth 1, that is 4 tuples, as the $cScore$ of $\epsilon_{A_1B_1}$ (i.e. $(x, 20)$) is larger than the upper bound of all the other combinations (i.e. $\epsilon^{max}_{A_1B_2} = 15$,$\epsilon^{max}_{A_2B_1} = 15$ and $\epsilon^{max}_{A_2B_2} = 10$). Thus, ETA is not an instance optimal algorithm. ■

Recall that ULA$^+$ always accesses no more objects than ULA (shown in Theorem 4). In the following proofs, for brevity, we focus on the optimality of only the ULA algorithm, which can be easily extended for ULA$^+$.

Following [10], we say that an algorithm makes *wild guesses* if it does random access to find the score of a tuple with ID x in some list before the algorithm has seen x under sorted access. For example, in Figure 1, we can see tuples whose IDs are $G04$ only at depth 3 under sorted and random accesses. But wild guesses can magically find $G04$ in the first step and obtain the corresponding scores. In other words, wild guesses can perform random jump on the lists and locate any tuple they want. In practice, we would not normally implement algorithms that make wild guesses. We prove the instance optimality of ULA (and ULA$^+$) algorithm, provided the size of each group is treated as a constant. This assumption is reasonable as it is mainly about assuming that the *schema* of the database is fixed.

Theorem 5 *Let $\mathbb{D}$ be the class of all databases. Let $\mathbb{A}$ be the class of all algorithms that correctly find top-k,m answers for every database and that do not make wild guesses. If the size of each group is treated as a constant, then ULA and ULA$^+$ are instance-optimal over $\mathbb{A}$ and $\mathbb{D}$.*

The next theorem shows that the upper bound of the optimality ratio of ULA is tight, provided the aggregation functions $\mathcal{F}_1$ and $\mathcal{F}_2$ are strictly monotone.

Theorem 6 *Assume that $\mathcal{F}_1$ and $\mathcal{F}_2$ are strictly monotonic functions. Let C_r and C_s denote the cost of one random access and one sorted access respectively. There is no deterministic algorithm that is instance-optimal for top-k,m problem, with optimality ratio less than $T + KC_r/C_s$, (which is the exact ratio of ULA), where $T = \sum_{i=1}^{n} g_i$, $K = \sum_{i \neq j}(g_i g_j)$, and g_i denotes the number of lists in group G_i.*

The detailed proofs of Theorem 5 and Theorem 6 can be founded in the conference version [26].

When we consider the scenarios when an algorithm makes *wild guesses*, unfortunately, our algorithms are not instance-optimal, but we can show that in this case *no* instance-optimal algorithm exists. Note that this appears a somewhat surprising finding, because the TA algorithm for top-k problems can guarantee instance optimality even under wild guesses for the data that satisfies the distinct property. In contrast, the ULA algorithm for top-k,m problem is not instance-optimal even for distinct data. The intuition for this disparity is that top-k problem needs to return the exact k objects, forcing all algorithms (including those with wild guesses) to go through the list to verify the results, but an algorithm for top-k,m search can correctly return k combinations without seeing their m objects by quickly locating a match instance to instantly boost the lower bound.

A1	A2	B1	B2
(1, 2n+1)	(x, n)	(y, n)	(2n+1, 2n+1)
.....	.....	.....	.....
(n, n+2)			(n+2, n+2)
(n+1, n+1)			(n+1, n+1)
(n+2, n)			(n, n)
.....			.....
(2n+1,1)			(1,1)

Fig. 8 Database for Theorem 7.

Theorem 7 *Let $\mathbb{D}$ be the class of all databases. Let $\mathbb{A}$ be the class of all algorithms (wild guesses are allowed) that correctly find top-k,m answers for every database. There is no deterministic algorithm that is instance-optimal over $\mathbb{A}$ and $\mathbb{D}$.*

Proof Let us consider a family of databases in Figure 8, assuming that $\mathcal{F}_1 = min$ and $\mathcal{F}_2 = sum$. Let $\mathscr{A}$ be an arbitrary deterministic algorithm in $\mathbb{A}$. Consider the top-$1,1$ query. It is easy to see that the expected number of accesses of algorithm $\mathscr{A}$ under this database is $n+5$ (i.e., $n+1$ sorted access to find the tuple (n+1, n+1) in list A_1 or B_2, and 3 sorted access to see the first tuples of the other lists, and 1 random access to find the tuples whose ID is "n+1" in the other list). Then $\mathscr{A}$ halts since $cScore(\epsilon_{A_1B_2}, 1) = n+1$ is larger than the upper bounds of all the other combinations (i.e. $\epsilon^{max}_{A_1B_1} = \epsilon^{max}_{A_2B_1} = \epsilon^{max}_{A_2B_2} = n$). However, there exists an algorithm $\mathscr{A}'$ that makes only 6 access (2 random accesses to find tuples (n+1, n+1) in list A_1 and B_2, and 4 sorted accesses to see the first objects of each list) to prove that $\{A_1, B_2\}$ is the final answer. Therefore, the optimality ratio may be arbitrarily large and the theorem follows. Note that the database constructed here satisfies the distinct property. ■

Finally, we consider the case (not so common in practice) when the number of attributes in each group is treated as a variable. While our algorithm is not instance-optimal in this case, we can show that *no* instance-optimal algorithm exists.

Theorem 8 *Let $\mathbb{D}$ be the class of all databases. Let $\mathbb{A}$ be the class of all algorithms that correctly find top-k,m answers for every database. If the number of elements in each group is treated as a variable, there is no deterministic algorithm that is instance-optimal over $\mathbb{A}$ and $\mathbb{D}$.*

Proof We prove by contradiction. Let us assume the existence of some optimal algorithm $\mathscr{A}$. Let $\mathcal{F}_1$ be *max* and $\mathcal{F}_2$ be *sum*. To answer a top-$1,1$ query, we construct a database $\mathscr{D}$ as follows (see Figure 9): there are two groups (G_1, G_2) and n attributes in each group. In G_1, there are n^2 distinct tuples in the first n lines and b_i in the depth $d \geq n$ (see the red box in G_1). In G_2, all objects in the first line are distinct with score 1, among them there is only one tuple whose ID

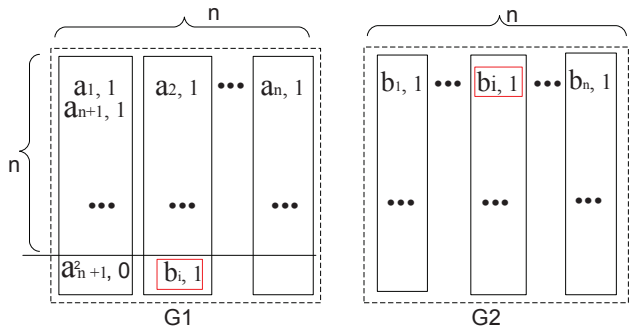


Fig. 9 Database for Theorem 8.

is b_i but the position of this tuple can be changed unless it is seen by the algorithm (see the red box in G_2). All Other tuples have score 0. Therefore, In $\mathscr{D}$, there is only one tuple in G_1 have the same ID with the tuple in G_2, that is, there is only one match instance, which consists of b_i. We now show, by an adversary argument, that the adversary can force $\mathscr{A}$ to have cost at least $\mathcal{O}(n^2)$ to see the match instance. But there exists another algorithm $\mathscr{A}'$ that, when executed over $\mathscr{D}$, runs only $\mathcal{O}(n)$ accesses. There are two cases.

Case 1: The algorithm $\mathscr{A}$ makes the sorted access on group G_1. The adversary can force $\mathscr{A}$ which tries to find the target object b_i, to access at least n^2 objects.

Case 2: The algorithm $\mathscr{A}$ makes the sorted access on group G_2 and does random access on group G_1. Whenever $\mathscr{A}$ does random access in group 1 for an object in the first line of group G_2, then the adversary assures that only the final random access find the target object b_i. Therefore the cost is at least $n^2 - 1$.

So in either case, the cost of $\mathscr{A}$ is at least $\mathcal{O}(n^2)$. But there exists another algorithm $\mathscr{A}'$, that accesses group 1 and directly finds the target object with cost $\mathcal{O}(n)$, which concludes the proof. ■

5.5 Theoretical analysis on the depth of accesses

In this subsection, we give a theoretical analysis on the average depth of accesses for our top-k,m algorithm. This quantitative analysis reveals the average performance of the algorithm.

We model our algorithm with the following procedure. Let $L_0, L_1, \ldots, L_M$ denote $M+1$ lists, each of which is a random permutation of set $\{1, \ldots, N\}$. For each list L_i, we use $L_i[1, \ldots, t]$ to denote the first t elements of L_i. Suppose we access each list sequentially in parallel, and stop at the Z-th access when there is a list L_i such that the intersection between the first Z elements of L_0 and the first Z elements of L_i has size at least K. We define Z to be the *depth of accesses*. In this section, we will prove that, if each L_i is a random permutation, with high probability, the depth of accesses is no more than $(\sqrt{\frac{e^6 NK}{M^{1/K}}} + K)$. In addition, we prove that the expected depth of accesses is $O(\sqrt{\frac{NK}{M^{1/K}}} + K)$.

Theorem 9 *Assume lists $L_0, L_1, \ldots, L_M$ are random permutations of set $\{1, \ldots, N\}$.*

1. *The probability that the depth of access is larger than $(\sqrt{\frac{e^6 NK}{M^{1/K}}} + K)$ is at most $e^{-\Omega(\frac{e^K}{\sqrt{K}})}$ for $M^{1/K} > e^6/3$, and at most $e^{-\Omega(MK)}$ for $M^{1/K} \le e^6/3$.*
2. *The expected depth of accesses is $O(\sqrt{\frac{NK}{M^{1/K}}} + K)$.*

Proof : Our basic idea is to derive an approximate distribution for each L_i, and take the minimum of these distributions. For $1 \le i \le M$, we define random variable Z_i to be the minimum index t such that the intersection between $L_0[1, \ldots, t]$ and $L_i[1, \ldots, t]$ has size at least K, i.e., $Z_i = \min_{1 \le t \le N} \{t \mid |\{1, \ldots, t\} \cap L_i[1, \ldots, t]| \ge K\}$. Let $Z = \min_{1 \le i \le M}\{Z_i\}$, then Z is the depth of accesses.

Let $F(t,l) = \frac{\binom{t}{l}\binom{N-t}{t-l}}{\binom{N}{t}}$. The cumulative distribution of each Z_i follows $\Pr[Z_i \ge t+1] = \sum_{l=0}^{k-1} F(t,l)$. Consequently, the cumulative distribution of Z follows $\Pr[Z_i \ge t+1] = \left(\sum_{l=0}^{k-1} F(t,l)\right)^M$, and the expected value of Z is equal to $E[Z] = \sum_{t=-1}^{N-1} \left(\sum_{l=0}^{k-1} F(t,l)\right)^M$.

If t is not too close to $\sqrt{NK}$, the two partial sums are dominated by $F(t,K)$. Therefore, we have (1) for $t \le \sqrt{NK/2}$, the summation $\sum_{l=K}^{t} F(t,l) \le 2F(t,K)$; and (2) for $t \ge \sqrt{2NK}$, we have $\sum_{l=0}^{K-1} F(t,l) \le 2F(t,K)$.

By using *Stirling's approximation* [1], we obtain an approximation for each individual $F(t,K)$. In particular, suppose $K \le N/20$. (1) For $t < \sqrt{3NK} + K$, we can lower bound $F(t,K)$ with $F(t,K) \ge \frac{1}{2\sqrt{2\pi K}} \left(\frac{e^{-5}(t-K)^2}{NK}\right)^K$; and (2) For $t \ge \sqrt{3NK} + K$, we can upper bound $F(t,K)$ with $F(t,K) \le \frac{1}{\sqrt{2\pi K}} e^{-\frac{(t-K)^2}{16N}}$.

(1) Proof of the high probability results. Suppose $M^{1/K} > e^6/3$. We define $T_1 = \sqrt{\frac{e^6 NK}{M^{1/K}}} + K$. With the help of the property of *Vandermonde's identity* [35], we have the following result:

$$\Pr[Z \ge T_1 + 1] \le e^{-M \cdot F(T_1, K)} = \exp\left(-\frac{e^K}{2\sqrt{2\pi K}}\right) \quad (1)$$

Now suppose $M^{1/K} \le e^6/3$ and define $T_2 = \sqrt{3NK} + K$, we have:

$$\Pr[Z \ge T_2 + 1] \le \left(\frac{2}{\sqrt{2\pi K}} e^{-\frac{(T_2 - K)^2}{16N}}\right)^M \le e^{-\frac{3MK}{16}}$$

Therefore, the high probability results hold.

(2) Proof of expected results. Let $x_0, x_1, \ldots, x_N$ denote a sequence of $N+1$ positive numbers. If for any $1 \le t-1 \le N$,

$x_{t-1}/x_t \geq e^{1/s}$ for some $s > 0$, then $\sum_{t=1}^{N} x_t \leq x_0(s+1)$. Since $x_{t-1}/x_t \geq e^{1/s}$ holds, we have $x_t \leq e^{-1/s}x_{t-1} \leq \ldots \leq (e^{-1/s})^t x_0$, and thus

$$\sum_{t=0}^{N} x_t \leq \sum_{t=0}^{N} (e^{-1/s})^t x_0 \leq x_0 \frac{1}{1-(1-\frac{1}{s+1})^{s/s}} = x_0(s+1)$$

According to the fact that $e^{-1} \leq (1+\frac{1}{s+1})^s$ for $s > 0$. We have

$$E[Z] = \sum_{t=-1}^{n-1} \Pr[Z \geq t+1] = \sum_{t=-1}^{n-1} \left(\sum_{l=0}^{K-1} F(t,l) \right)^M$$

$$= \sum_{t=-1}^{T_1-1} \Pr[Z \geq t+1] + \sum_{t=T_1}^{T_2-1} \Pr[Z \geq t+1] + \sum_{t=T_2}^{T_3-1} \Pr[Z \geq t+1],$$

where $T_1 = \sqrt{e^3 NK/M^{1/K}} + K$, $T_2 = \sqrt{3NK} + K$ and $T_3 = \frac{N+K}{2}$.

The first summation can be bounded as:

$$\sum_{t=-1}^{T_1-1} \Pr[Z \geq t+1] \leq T_1 + 1 = \sqrt{\frac{e^3 NK}{M^{1/K}}} + K + 1 \quad (2)$$

The second summation can be bounded as:

$$\sum_{t=T_1}^{T_2-1} \Pr[Z \geq t+1] \leq \sum_{t=T_1}^{T_2-1} x_t \leq o\left(\sqrt{\frac{NK}{M^{1/K}}}\right) \quad (3)$$

The third summation can be bounded as:

$$\sum_{t=T_2}^{T_3-1} \Pr[Z \geq t+1] \leq \sum_{t=T_2}^{T_3-1} F(t,K)^M \leq \sum_{t=T_2}^{T_3-1} e^{-\frac{M(t-K)^2}{16N}}$$

$$\leq \sqrt{\frac{64N}{3M^2K}} + 1 = o\left(\sqrt{\frac{NK}{M^{1/K}}}\right) \quad (4)$$

Combining Equation 2, Equation 3 and Equation 4, we have $E[Z] = O\left(\sqrt{\frac{NK}{M^{1/K}}} + K\right)$. ■

Remark. Both the probability bounds and the expectation results are necessary. For large K, the probability results ensure that the depth of accesses is $O(\sqrt{\frac{NK}{M^{1/K}}} + K)$ with extremely high probability. However, when K is a constant (in particular when $K = 1$), this probability becomes constant, and the expected result is more useful in this case. We also notice that when $M^{1/K}$ is a constant, the depth of accesses is $O(\sqrt{NK})$, which is the same bound achieved in [10]. However, when $M^{1/K}$ becomes large, our bound is superior in both expectation and rate of convergence, since the probability of the depth of accesses exceeds a constant times the expectation is double exponentially small.

Algorithm 3: The ENRA algorithm

(i) Initial an empty set S_ϵ for each combination ϵ.
(ii)Do sorted access in parallel to each sorted list.
(iii) For combination ϵ,
  (1) [*growing phase*]: compute current threshold value $\mathcal{T}^\epsilon$, add seen objects into S_ϵ, and compute $\mathcal{I}^{min}$ and $\mathcal{I}^{max}$ for them;
  (2)[*shrinking phase*]: If there are at least m objects in S_ϵ whose $\mathcal{I}^{min} \geq \mathcal{T}^\epsilon$, stop adding new objects to S_ϵ but only update the scores for objects in S_ϵ;
  (3) [*scanning phase*]: If there are at least m objects in S_ϵ whose $\mathcal{I}^{min}$ are no less than the $\mathcal{I}^{max}$ of other objects, remove all the objects except the m objects in S_ϵ and continually updating the scores for the m objects until they get m match instances.
(iv) For every combination, if the $tScore$ of the top-m match instance could be calculated, ENRA halts, otherwise go to step
(v) Let Y be a set containing the k combinations whose $cScore$ are larger than that of others. Output Y.

6 Top-k,m algorithms with no random access

In real life scenarios the cost of random access (RA) might be one to two order of magnitudes higher than that of sorted access (SA). More precise, for very large index lists with millions of entries that span multiple disk tracks, the resulting random access cost is $50 \sim 50,000$ times higher than the cost of a sorted access [2]. Further more, random accesses are not allowed due to the property of data sources. For example: (i) the ranked lists are input as stream data, then tuples could only be read one by one and the unseen tuples could not be randomly accessed by its key; and (ii) the ranked lists are provided by a search engine, and thus it does not seem to be a way to ask a major search engine on the Web for its internal score on some document of our choice under a query. Therefore, motivated by the previous examples, we proceed to propose new algorithms that make no random accesses for the top-k,m problem.

The main challenge brought by *no* random accesses is that a seen tuple could not obtain its corresponding match instance immediately. More precisely, for a combination ϵ, if random accesses are not allowed, then a match instance $\mathcal{I} = \{\tau_1, \ldots, \tau_n\}$ could only be obtained after $\forall \tau_i$, $i \in [1,n]$ has been seen by sorted access. In the worst case, we need to scan the whole lists to find the match instance. For example, consider the match instance $\{(G09, 2.06), (G09, 1.98), (G09, 7.10)\}$ for combination $\{F_1C_1G_2\}$ in Figure 1(a). To get the match instance, we need to scan the whole list in F_1 and C_1 by sorted accesses. However, if random accesses are allowed, once we access the tuple $(G09, 7.10)$ in G_2, we could get the match instance by random accesses immediately.

6.1 Baseline algorithm with *no* random accesses: ENRA

The well-known instance-optimal top-k algorithm, i.e. NRA, that does not make random accesses proposed by [10] works

A_1	A_2
(x,20)	(1, 10)
.....	.....
	(n, 10)
	(n+1, 10)
	(n+2, 1)
	.....
	(2n+1,1)

B_1	B_2
(x,20)	(2n+1, 10)
.....	.....
	(n+2, 10)
	(n+1, 10)
	(n, 1)
	.....
	(1,1)

Fig. 10 Sub-optimality of ENRA.

as follows:(i) at each depth, compute the upper and lower bounds for each seen object (by sorted accesses); and (ii) NRA halts, if (1) there are at least k objects have been seen, and (2) there are at least k objects whose lower bounds are no less than the upper bounds of the other objects.

Further, [27] proposes two phases (i.e. *growing phase* and *shrinking phase*) to optimize NRA by minimizing the objects stored in memory. In *growing phase*, all the tuples seen by sorted accesses must be stored since all of them have the chance to be a result. If there exist k objects whose lower bounds are greater than the current threshold value, then change to *shrinking phase*, which means unseen objects can never be answered, so it is not necessary to store those objects.

To answer top-k,m problem with *no* random accesses, one method is to extend NRA (called ENRA) to compute top-m match instances for each combination. However, this straightforward extension needs to fix a core problem, that is, NRA only returns the top-m object IDs without scores, and thus we cannot calculate *cScore* for each combination. Therefore, we extend NRA by performing an additional phase, called *scanning phase*, to do further accesses for those top-m objects to get their $tScore$'s. Note that, in the *scanning phase*, we only update the scores for those top-m objects.

In the ENRA algorithm (see Algorithm 3), we first define the upper and lower bounds for a match instance $\mathcal{I}$. Considering a combination ϵ=$\{e_1, \ldots, e_n\}$, let x_i denote the current score of the tuple in the list L_i at depth d, where L_i corresponds to an attribute e_i. For a match instance $\mathcal{I} \in \epsilon$, assume we have accessed m tuples $\tau_1, \ldots, \tau_m$ for $\mathcal{I}$, where $m < n$ and $\rho(\tau_1) = \rho(\tau_2) =, \ldots, \rho(\tau_m)$. It means that only partial match instance has been accessed and the *tSocre* of $\mathcal{I}$ could not be calculated now. However, we define the upper bound $\mathcal{I}^{max}$ and lower bound $\mathcal{I}^{min}$ for $\mathcal{I} \in \epsilon$ as follows:

$$\mathcal{I}^{max} = \mathcal{F}_1(\sigma(\tau_1), \ldots, \sigma(\tau_m), x_{m+1}, \ldots, x_n)$$

$$\mathcal{I}^{min} = \mathcal{F}_1(\sigma(\tau_1), \ldots, \sigma(\tau_m), \underbrace{0, \ldots, 0}_{n-m})$$

We describe the ENRA algorithm in Algorithm 3, which is naturally extended from the NRA algorithm in [10] by adding the scanning phase.

Example 6 We employ the data in Figure 1 to show how ENRA works. Assume that $\mathcal{F}_1$ and $\mathcal{F}_2$ are sum, and the query is top-$1, 1$. Note that only sorted accesses are allowed. Consider the combination $\epsilon = \{F_1C_2G_1\}$, during the *growing phase* all the accessed tuples (e.g, (G01,9.31)) are stored. At depth 4, ϵ changes to *shrinking phase*, since there exists at least one object whose lower bounds is bigger than the threshold value $\mathcal{T}^{\epsilon}$ (12.06), e.g., $\mathcal{I}^{min}_{G01} = 13.12$ and $\mathcal{I}^{min}_{G03} = 15.06$. Then at depth 7, ϵ transforms to the *scanning phase* and get the top-1 match instance with $tScore$ 16.5. Finally, ENRA halts at depth 7 and the answer is $\{F_2C_1G_1\}$ with $cScore$ 21.51. □

We show that ENRA is not instance optimality using the following claim.

Lemma 7 *Given a class of databases $\mathbb{D}$, and a class of algorithms $\mathbb{A}$ that correctly find top-k,m answers for every database and do not make random accesses. ENRA is not instance optimal among $\mathbb{A}$ for $\mathbb{D}$.*

Proof Consider the example in Figure 10. Assume both $\mathcal{F}_1$ and $\mathcal{F}_2$ are sum. ENRA halts by depth $n + 1$, since it needs to see the object "n+1" in both lists A_2 and B_2 to obtain the top-1 match instance for combination $\epsilon_{A_2B_2}$, that is, ENRA needs to obtain the exact top-m match instances for each combination. However, there exists one algorithm $\mathscr{A} \in \mathbb{A}$, which only access 4 objects (the first object of each list), then $\mathscr{A}$ halts. Because the current score of combination $\epsilon_{A_1B_1}$ $(40 = 20 + 20)$ is larger than all the possible maximal scores of all the other combinations ($\epsilon^{max}_{A_1B_2} = 30$, $\epsilon^{max}_{A_2B_1} = 30$, $\epsilon^{max}_{A_2B_2} = 20$), as desired. ■

6.2 No random access algorithm: NULA

The ENRA algorithm needs to compute the exact top-m match instances for each combination, which is time-consuming and reads more tuples than needed. Therefore, we propose a novel efficient top-k,m problem with *no* random accesses. We follow the line of ULA algorithm, i.e., calculate the upper and lower bounds for each combination instead of the exact $cScore$. However, as we discussed above, without random accesses, some objects only obtain partial match instances. Therefore, we need to redefine the upper and lower bounds of the combination by considering the scores of the partial match instances.

Given a combination ϵ, assume m' distinct tuples have been seen at depth d. Among which $m_1 \leq m'$ distinct tuples have already been match instances (i.e., $\mathcal{I}_1, \ldots, \mathcal{I}_{m_1}$), and $m_2 = m' - m_1$ distinct tuples only obtain the partial match instances (their upper and lower bounds are $\mathcal{I}^{max}_i$ and $\mathcal{I}^{min}_i$ $i \in [1, m_2]$). Then we define the upper and lower bounds for combinations in top-k,m problem with *no* random accesses as follows:

Upper bound ϵ^{max}: The upper bound of ϵ would be computed as follows:

$$\begin{cases} \mathcal{F}_2(\underbrace{tScore(\mathcal{I}_1^\epsilon),...,tScore(\mathcal{I}_{m_1}^\epsilon)}_{m_1},\underbrace{\mathcal{I}_1^{max},...,\mathcal{I}_{m_2}^{max}}_{m_2},\underbrace{\mathcal{T}^\epsilon,...,\mathcal{T}^\epsilon}_{m-m'}) \\ if\ m' < m \\ \mathcal{F}_2(max\{\underbrace{tScore(\mathcal{I}_i^\epsilon),...,tScore(\mathcal{I}_j^\epsilon),\mathcal{I}_i^{max},...,\mathcal{I}_j^{max}}_{m}\}) \\ if\ m' \geq m \end{cases}$$

If $m' = m_1 + m_2 < m$, ϵ^{max} is padded with $m - m'$ $\mathcal{T}^\epsilon$'s, otherwise, meaning that the exact top-m match instances must be among the $m_1 + m_2$ match instances (including the potential match instances).

Lower bound ϵ^{min}: The lower bound would be computed as follow:

$$\begin{cases} \mathcal{F}_2(\underbrace{tScore(\mathcal{I}_1^\epsilon),\cdots,tScore(\mathcal{I}_{m_1}^\epsilon)}_{m_1},\underbrace{\mathcal{I}_1^{max},\cdots,\mathcal{I}_{m_2}^{max}}_{m_2},\underbrace{0,\cdots,0}_{m-m'}) \\ if\ m' < m \\ \mathcal{F}_2(max\{\underbrace{tScore(\mathcal{I}_i^\epsilon),\cdots,tScore(\mathcal{I}_j^\epsilon),\mathcal{I}_i^{max},\cdots,\mathcal{I}_j^{max}}_{m}\}) \\ if\ m' \geq m \end{cases}$$

If $m' = m_1 + m_2 < m$, ϵ^{max} is padded with $m - m'$ 0's, otherwise, meaning that the exact top-m match instances must be among the $m_1 + m_2$ match instances (including the potential match instances).

Example 7 We employ the data in Figure 1 again to show how to compute ϵ^{max} and ϵ^{min}. Assume that $\mathcal{F}_1$ and $\mathcal{F}_2$ are sum, and the query is top-$1, 2$. Consider combination $\epsilon = \{F_2C_1G_1\}$, at depth 2, the threshold value is $20.27 = 8.07 + 6.01 + 6.19$. In addition, we access 4 distinct tuple ID's, i.e., G02 ($tScore(\mathcal{I}_{G02}) = 21.51$), G08($\mathcal{I}_{G08}^{max} = 20.27$ and $\mathcal{I}_{G08}^{min} = 8.07$), G05($\mathcal{I}_{G05}^{max} = 21.47$ and $\mathcal{I}_{G05}^{min} = 7.21$) and G03($\mathcal{I}_{G03}^{max} = 20.27$ and $\mathcal{I}_{G03}^{min} = 6.19$). Thus, the upper bound $\epsilon^{max} = tScore(\mathcal{I}_{G02}) + \mathcal{I}_{G05}^{max} = 21.51 + 21.47 = 42.98$ and the lower bound is $\epsilon^{min} = tScore(\mathcal{I}_{G02}) + \mathcal{I}_{G08}^{min} = 21.51 + 8.07 = 29.58$. □

We would show that the $cScore$ of each combination is distributed in ϵ^{min} and ϵ^{max}. (i) For each seen object that obtains the complete match instance, then $\mathcal{I}^{min} = tScore = \mathcal{I}^{max}$ holds. (ii) For each seen object that only get the partial match instance, then the maximal $tScore$ is no more than $\mathcal{I}^{max}$, since the unseen scores in list L_i is no more than x_i(x_i denotes the current score of the tuple in list L_i at depth d), and the minimal $tSocre$ is no less than $\mathcal{I}^{min}$. (iii) For the unseen object, the $tScore$ is no more than $\mathcal{T}^\epsilon$ and no less than 0. By the definition of ϵ^{min} and ϵ^{max}, it is easy to see that the $cScore$ of each combination ϵ should distribute in the scope of $[\epsilon^{min},\epsilon^{max}]$.

We are now ready to show a novel algorithm named **NULA**, which would stop earlier than the ENRA algorithm by exploring the relation of the upper and lower bounds of combinations. See Algorithm 4. Note that, the *growing* phase and *shrinking* phase could be applied here, but the *scanning phase* is unnecessary, as we can avoid to compute the exact scores for top-m instances.

Algorithm 4: The NULA algorithm

(1) Initial an empty set S_ϵ for each combination ϵ.
(2)Do sorted access in parallel to each of the lists by sorted accesses.
(3) For each unterminated combination ϵ, (i) if ϵ is in *growing* phase, then add the current objects into S_ϵ and update $\mathcal{I}^{min}$ for each object; (ii) check whether ϵ is in *shrinking phase* now; (iii) if ϵ is in *shrinking phase*, update $\mathcal{I}^{min}$ and $\mathcal{I}^{max}$ for each object in S_ϵ and stop adding new objects to S_ϵ; (iv) compute ϵ^{min} and ϵ^{max}, and (v) check if ϵ is terminated now (the same terminate condition as that in ULA algorithm).
(4) If there are k combinations which meet the hit-condition, then the algorithm halts. Otherwise, go to step 1.
(5) Let Y be a set containing the k combinations when NULA halts. Output Y.

Theorem 10 *If the aggregation functions $\mathcal{F}_1$ and $\mathcal{F}_2$ are monotone, then NULA correctly finds the top-k,m answers.*

Proof Let Y be the results set, we claim that the $cScore$ of each combination $\epsilon \in Y$ is larger than that of $\xi \notin Y$. In NULA, the $cScore$ of combination $\epsilon \in Y$ is no less than ϵ^{min}, and the $cScore$ of combination $\xi \notin Y$ is no more than ξ^{max}. Since for each $\epsilon \in Y$ and $\xi \notin Y$, the $\epsilon^{min} \geq \xi^{max}$, Therefore, $cScore(\epsilon, m) \geq cScore(\xi, m)$ holds, as desired. ■

Example 8 We continue the example in Figure 1 again to present how NULA works. Assume that $\mathcal{F}_1$ and $\mathcal{F}_2$ are sum, and the query is top-$1, 1$. Let $\mathbb{E}$ be the collection of combinations. At depth 4, the lower bound of combination $\{F_2C_1G_1\}$ is no less than the upper bound of the other combinations. More precisely, $\epsilon_{F_2C_1G_1}^{min} = 21.51 \geq max\{\xi^{max} | \xi \in \mathbb{E}/\epsilon\}$. Therefore, NULA outputs $\{F_2C_1G_1\}$ as the answer. To demonstrate the efficiency of NULA, we show the following data: ENRA halts at depth 7 and accesses 42 tuples, while NULA halts at depth 4 and accesses 24 tuples. □

6.3 Optimized no random access algorithm: NULA$^+$

In the following, we aim to propose an optimized top-k,m algorithm with *no* random accesses, named NULA$^+$, which improves NULA by avoiding sorted access on some lists.

We observe that the optimizations (Lemma 5 and Lemma 6) we proposed above could not be applied in NULA$^+$, since random accesses are forbidden here. However, Lemma 4 could be utilized with minor modification.

Lemma 8 *During query processing, given a list L, if all the combinations involving L are terminated, then we do not need to perform sorted accesses upon the list L any longer.*

Proof The proof is similar to the proof of Lemma 4. That is, if a combination ϵ is terminated, then adding new match instances is useless to ϵ. ■

The NULA$^+$ applies Lemma 8 upon NULA algorithm to reduce the number of sorted accesses. At each depth d, as shown in Line 2 of Algorithm 4, NULA$^+$ additionally checks whether a list meets the condition in Lemma 8, if yes, stop accessing the list.

6.4 Optimality properties of NULA and NULA$^+$

Theorem 11 *Let $\mathbb{D}$ be the class of all databases. Let $\mathbb{A}$ be the class of all algorithms that correctly find top-k,m answers for every database and that do not make random accesses. NULA and NULA$^+$ are instance optimal over $\mathbb{A}$ and $\mathbb{D}$.*

Proof Let C_s denotes the cost of one sorted access. Assume that $\mathscr{A} \in \mathbb{A}$, and $\mathscr{A}$ runs over database $\mathscr{D} \in \mathbb{D}$. Assume that each object can be a match instance for each combination. We claim that if NULA halts at depth $d+m$, then $\mathscr{A}$ could not halt earlier than $d+1$. Then the cost of $\mathscr{A}$ is at least $(d+1+\sum_{i=1}^{n} g_i - 1)C_s$, where g_i denote the number of elements in the group G_i. And the cost of NULA is at most $(d+m)(\sum_{i=1}^{n} g_i)C_s$, which is $d(\sum_{i=1}^{n} g_i)C_s$ plus an additive constant of $m(\sum_{i=1}^{n} g_i)C_s$. Let $T = \sum_{i=1}^{n} g_i$, then the optimality ratio of NULA is at most $\frac{dTC_s}{dC_s} = T$. Suppose the contrary. Consider the $\mathscr{A}$ halts by depth d, at this time, NULA does not halts, that is, there are less than k distinct combinations whose lower bound are larger than the upper bound of the other combinations. Let Y be the results set output by $\mathscr{A}$, then there is a combination $\epsilon \in Y$ and another combination $\xi \notin Y$, which does not share lists with ϵ. Then $\epsilon_{(d)}^{min} < \xi_{(d)}^{max}$ at depth d. We construct a database $\mathscr{D}'$ where $\mathscr{A}$ errs as follows. Database $\mathscr{D}'$ is just like to $\mathscr{D}$ up to depth d. For each list $L_i \in \epsilon$ (i.e., the element e associated with L_i belongs to ϵ), we assign $U_1,\ldots,U_m$ with score 0 and also 0 to the other objects below U_m in L_i. And assign $V_1,\ldots,V_m$ with grade $\underline{x}_i$ to each list $L_j \in \xi$. Therefore, we would get the $cScore(\epsilon, m)$ and $cScore(\xi, m)$ at depth $d+m$. We have $cScore(\epsilon, m) = \epsilon_{(d)}^{min}$ and $cScore(\xi, m) = \xi_{(d)}^{max}$, then $cScore(\epsilon, m) < cScore(\xi, m)$, since $\epsilon_{(d)}^{min} < \xi_{(d)}^{max}$, $\mathscr{A}$ errs, as desired. ■

Theorem 12 *Assume that $\mathcal{F}_1$ and $\mathcal{F}_2$ are strict aggregation functions. Let $\mathbb{D}$ be the class of all databases, Let $\mathbb{A}$ be the class of all the deterministic algorithm that correctly find the top-k,m combinations for every database and that do not make random access. There is no deterministic algorithm that is instance optimal over $\mathbb{A}$ and $\mathbb{D}$, with optimality ratio less than T, (which is the exact ratio of NULA), where $T = \sum_{i=1}^{n} g_i$.*

Proof We assume without loss of generality that $k = 1$ and $m = 1$. We restrict our attention to a subfamily $\mathbb{D}'$ of $\mathbb{D}$, by making use of a (large) positive integer parameter d ($d > T$, where $T = \sum_{i=1}^{n} g_i$). The family $\mathbb{D}'$ contains every database of the following form.

In every list, the top d scores are 1, and the remaining scores are 0. No match instance is in the top T. There is only one object τ that has score 1 in all of the lists of one combination, and it is in the top d of one list and in the top T of the other lists.

Let $\mathscr{A}$ be an arbitrary deterministic algorithm. We now show, by an adversary argument, that the adversary can force $\mathscr{A}$ to have the cost at least dT on some database in $\mathscr{D}$. The idea is that the adversary dynamically adjusts the database as each query comes in from $\mathscr{A}$, in such a way as to evade allowing $\mathscr{A}$ to determine the top element until as late as possible.

It is clear that the sorted access cost of $\mathscr{A}$ on this resulting database is at least dTC_s. However, there is an algorithm in $\mathbb{A}$ that makes at most d sorted access to one list and T sorted accesses to each of the remaining lists, that is, at most d+$(T-1)T$ sorted accesses and the cost is d+$(T-1)T$ C_s. By choosing d sufficiently large, the ratio $\frac{dTC_s}{d+(T-1)TC_s}$ can be made as close as desired to T, Then the theorem follows. ■

7 Approximate Top-k,m algorithms

In some query processing environments, e.g., Online analytical processing (OLAP), obtaining exact Top-k,m query answers may be overwhelming to database engines because of the interactive nature of such environments, and the large volume of data they store. Such environments tend to sacrifice the accuracy of query answers in favor of performance. Therefore, it is acceptable for a top-k,m query to return approximate answers.

In this section, we show how to extend our exact top-k,m algorithms to accommodate this approximate setting. The approximate answers should not be arbitrarily far from the exact answers. They are expected to be associated with guarantees indicating how far they are from the exact answers. In our approximate top-k, m algorithms, the approximate ratio (i.e., the threshold of the largest distance between approximate answers and the exact answers) is given by users in order to make the approximate algorithms more flexible. More precisely, given an approximate ratio $\theta > 1$, for any answer combination ϵ returned by our approximate top-k, m algorithms and any other combination ξ not in the answer set, the condition of final scores $\theta \cdot cScore(\epsilon, m) \geq cScore(\xi, m)$ is always hold.

Approximate baseline algorithms To extend the baseline method ETA (Section 5.1) and ENRA (Section 6.1) to the corresponding approximate versions. The only modification is to change the output condition from $cScore(\epsilon, m) \geq cScore(\xi, m)$ to $\theta \cdot cScore(\epsilon, m) \geq cScore(\xi, m)$, where $\epsilon \in Y$ and $\xi \notin Y$ and Y is the answer set. The approximate ETA and ENRA are named as ETA$_\theta$ and ENRA$_\theta$ respectively. The correctness of ETA$_\theta$ and ENRA$_\theta$ is provided in Theorem 13.

Approximate top-k, m algorithms To modify the top-k, m algorithms with lower and upper bounds, i.e., ULA (Section 5.2), ULA$^+$ (Section 5.3), NULA (Section 6.2) and NULA$^+$ (Section 6.3), to accommodate the approximate environment, we need to change the *hit-condition* and the *drop-condition* as follows:

- The condition of hit-condition is changed from $\epsilon^{min} \geq max(\xi_i^{max} | 1 \leq i \leq \mathbb{N} - k)$ to $\epsilon^{min} \geq max(\frac{\xi_i^{max}}{\theta} | 1 \leq i \leq \mathbb{N} - k)$. It indicates that if a combination ϵ is an approximate answer, then its lower bound should be at least greater than the $\frac{1}{\theta}$ of the upper bounds of other $\mathbb{N}-k$ combinations.
- The condition of drop-condition is modified from $\xi^{max} < min(\epsilon_i^{min} | 1 \leq i \leq k)$ to $\frac{\xi^{max}}{\theta} < min(\epsilon_i^{min} | 1 \leq i \leq k)$. It means that if a combination ξ is not an approximate answer, then the $\frac{1}{\theta}$ of its upper bound is smaller than the lower bounds of k other combinations.

By using the new hit-condition and drop-condition, we are able to provide approximate top-k, m algorithms with result guarantees. The corresponding approximate top-k,m algorithms are named as ULA$_\theta$, ULA$_\theta{}^+$, NULA$_\theta$ and NULA$_\theta{}^+$ respectively. The correctness of these algorithms is shown in Theorem 13.

Theorem 13 *Assume $\theta > 1$ and the aggregation functions $\mathcal{F}_1$ and $\mathcal{F}_2$ are monotone, then the family of our top-k,m approximate algorithms correctly finds the top-k,m answers.*

Proof Let Y be the results set, we claim that the θ $cScore$ of each combination $\epsilon \in Y$ is larger than the $cScore$ of $\xi \notin Y$. We discuss the correctness of our algorithms in two classes, i.e., baseline methods and ULA-based approaches.

(1) *Baseline methods: ETA$_\theta$, ENRA$_\theta$*. They compute the exact $cScore$ for each combination. Thus, it is easy to choose the combinations satisfying $\theta \cdot cScore(\epsilon, m) \geq cScore(\xi, m)$ into the result set Y, as desired.

(2) *ULA-based approaches: ULA$_\theta$, ULA$^+{}_\theta$, NULA$_\theta$ and NULA$^+{}_\theta$*. The $\theta \cdot cScore(\epsilon, m)$ of combination $\epsilon \in Y$ is no less than $\theta \cdot \epsilon^{min}$, and the $cScore(\xi, m)$ of combination $\xi \notin Y$ is no more than ξ^{max}. Since the modified hit-condition is $\epsilon^{min} \geq \max(\frac{\xi_i^{max}}{\theta} \mid 1 \leq i \leq \mathbb{N} - k)$, which means the $\theta \cdot \epsilon^{min} \geq \xi^{max}$ holds for the combinations $\epsilon \in Y$ and $\xi \notin Y$. Therefore, we have $\theta \cdot cScore(\epsilon, m) \geq cScore(\xi, m)$, as desired. ■

Theorem 14 *Assume $\theta > 1$ and the aggregation functions $\mathcal{F}_1$ and $\mathcal{F}_2$ are monotone. Let $\mathbb{D}$ be the class of all databases. Let $\mathbb{A}$ be the class of all algorithms that correctly find a θ-approximation to the top-k,m answers for every database and that do not make wild guesses. Then ULA$_\theta$ and ULA$_\theta^+$ (also NULA$_\theta$ and NULA$_\theta^+$) are instance optimal over $\mathbb{A}$ and $\mathbb{D}$.*

Proof We prove this theorem by following the similar way of Theorem 5. The core task is to show that if an optimal algorithm $\mathscr{A} \in \mathbb{A}$ over every database $\mathscr{D} \in \mathbb{D}$ halts by sorted access at most up to depth d, then our algorithms halt on $\mathscr{D}$ by sorted access at most up to depth $d + m$.

If algorithm $\mathscr{A}$ meets one of the following two conditions when it halts: (i) $\mathscr{A}$ has seen the exact top-m match instances for each combination; (ii) $\mathscr{A}$ has not seen the exact top-m match instances for each combination, but $\theta \cdot \epsilon^{\min} \geq \xi^{\max}$ holds for any combination $\epsilon \in Y$ and combination $\xi \notin Y$. Then our algorithms also halt by depth d<$d + m$, as desired.

Otherwise, assume there exists one combination $\epsilon \in Y$ and one combination $\xi \notin Y$ such that $\theta \cdot \epsilon^{\min} < \xi^{\max}$. At this point, our algorithms cannot stop immediately. But since $\mathscr{A}$ is correct without seeing the remaining tuples after depth d, we can prove that the combinations in Y have an important property, i.e., $\theta \cdot mScore(\epsilon, m) \geq hScore(\xi, m)$, where $mScore(\epsilon, m)$ and $hScore(\xi, m)$ are computed as follows:

$$mScore(\epsilon, m) = \mathcal{F}_2(tScore(\mathcal{I}_1^\epsilon), \ldots, tScore(\mathcal{I}_{m'}^\epsilon), \underbrace{\omega, \ldots, \omega}_{m-m'})$$

where $\omega = \mathcal{F}_1(\sigma_1^\epsilon, \ldots, \sigma_{|\epsilon|}^\epsilon)$ denote the possible minimal $tScore$ (Assume that $\mathscr{A}$ has seen m' ($m' \leq m$) match instances for ϵ and let σ_i^ϵ denote the seen minimal score (under sorted or random accesses) in L_i at depth d).

$$hScore(\xi, m) = \mathcal{F}_2(tScore(\mathcal{I}_1^\xi), \ldots, tScore(\mathcal{I}_{m''}^\xi), \underbrace{\varphi, \ldots, \varphi}_{m-m''})$$

where $\varphi = \mathcal{F}_1(\lambda_1^\xi, \ldots, \lambda_{|\xi|}^\xi)$ denote the possible maximal $tScore$ (Assume that $\mathscr{A}$ has seen m'' ($m'' \leq m$) match instances for ξ. Let λ_i^ξ denote the unseen possible maximal score ($\lambda_i^\xi \leq \sigma(\tau)$) below τ in list i by depth $d + (m - m'')$ of ξ).

Then we construct a database $\mathscr{D}'$ the same to the database in Theorem 5 to demonstrate that all the combinations in Y satisfy the condition $\theta \cdot mScore(\epsilon, m) \geq hScore(\xi, m)$. Therefore by depth $d + m$, our algorithms would get at least m match instances, and the θ times of the lower bound of ϵ is no less than $\theta \cdot mScore$, and the upper bound of ξ is no more than $hScore$. Thus, by depth $d + m$, $\theta \cdot \epsilon^{\min} \geq \xi^{\max}$. Therefore, our algorithms halt by depth $d + m$, as desired. ■

8 Case studies on real applications

In this section, we introduce a case study for top-k,m queries. In the preliminary version [26], we have studied how to rewrite an XML keyword query by answering a top-k,m query with random access. Here, we show another example on how to use top-k,m queries with no random accesses to perform online query rewriting in a biomedical search engine PubMed[5].

Given a search query, PubMed returns a list of documents containing the keywords in the query. Usually the size of answers is large. Returning such big results to users in one page makes no sense. So PubMed only returns the top-k, e.g., $k = 20$, to users. If users want more answers, then users can issue another request by clicking "Next" Button. In this way: a random access is not supported in PubMed. Therefore, only no random accesses top-k,m algorithms can be applied here to provide keyword refinement. Therefore, PubMed is a typical application for top-k,m queries with *no* random accesses.

Given a set of keywords (called terms or words interchangeably), we study how to automatically rewrite the keywords to provide users better and more relevant search results on PubMed, as in real applications users' input may not have answers or the answers are not good. We assume that there exists a table containing simple rules in the form of $A \rightarrow B$, where A and B are two strings, which means A and B refer to the same entity. E.g., "Achlorhydria"→"Hypochlorhydria", and "Hypopotassemia"→"Hypokalemia". These rules can be obtained from existing dictionaries, say medical subject headings (MeSH) [6]. In Section 9, we will list more example rules that are collected by domain experts and can be searched in MeSH.

Now we illustrate how to perform a top-k, m search in biomedical citations query refinement. Given a query $q = \{q_1, \ldots, q_n\}$, we scan all keywords sequentially and perform substring match by rules to generate groups. Assuming q="Achlorhydria, Hypopotassemia", we have two groups, i.e., G_1={Achlorhydria, Hypochlorhydria} and G_2= {Hypopotassemia, Hypokalemia} by using the rules.

Further, to construct sorted lists for each element in groups, note that, here only sorted accesses are supported by PubMed. For each keyword w, PubMed returns a sorted list including all the citations containing w. We use the following example to illustrate how the NULA algorithm works for query rewriting on an online biomedical database.

Example 9 Consider the example in Figure 11 in order to illustrate how NULA (our top-k, m algorithms with only sorted accesses) find the top-1 refined query according to top-2 highest scores of a citation, i.e., top-$1, 2$ query. Here we perform sorted accesses to the first tuple in each list, then compute their upper and lower bounds according to the formulas in Section 6. That is $\epsilon^{max}_{A_1B_1}$=2.82, $\epsilon^{min}_{A_1B_1}$=1.41, $\epsilon^{max}_{A_1B_2}$=3.86, $\epsilon^{min}_{A_1B_2}$=1.93, $\epsilon^{max}_{A_2B_1}$=1.80, $\epsilon^{min}_{A_2B_1}$=0.90, and also $\epsilon^{max}_{A_2B_2}$=2.84, $\epsilon^{min}_{A_2B_2}$=1.42. Then, $\epsilon_{A_2B_1}$ can be pruned due to the fact that $\epsilon^{max}_{A_2B_1} < \epsilon^{min}_{A_1B_2}$. Then we perform sorted access to second tuples in each list and update the upper and lower bounds for remaining combinations. That is $\epsilon^{max}_{A_1B_1}$=2.70, $\epsilon^{min}_{A_1B_1}$=1.91, $\epsilon^{max}_{A_1B_2}$=3.77, $\epsilon^{min}_{A_1B_2}$=2.85, and also $\epsilon^{max}_{A_2B_2}$=2.75, $\epsilon^{min}_{A_2B_2}$=1.86. Then A_1B_2 can be guaranteed to be the top-1 combination with highest top-2 matching instances, because $\epsilon^{min}_{A_1B_2}$ is larger than both $\epsilon^{max}_{A_1B_1}$ and $\epsilon^{max}_{A_2B_2}$. Therefore, "*Achlorhydria*, *Hypokalemia*" is returned as a refined query for the original query "*Hypochlorhydria*, *Hypokalemia*". □

A1	A2
Achlorhydria	Hypochlorhydria
(D06, 0.99)	(D10, 0.88)
(D05, 0.92)	(D03, 0.81)
(D03, 0.88)	(D08, 0.75)
(D10, 0.80)	(D09, 0.72)
(D09, 0.68)	(D04, 0.60)
(D02, 0.47)	(D05, 0.48)
(D07, 0.39)	(D01, 0.36)
(D08, 0.35)	(D06, 0.21)
(D04, 0.31)	(D02, 0.19)
.....	.....
(D01, 0.11)	(D07, 0.08)

B1	B2
Hypopotassemia	Hypokalemia
(D02, 0.92)	**(D05, 0.94)**
(D03, 0.77)	(D08, 0.92)
(D08, 0.72)	(D06, 0.92)
(D09, 0.67)	(D04, 0.86)
(D10, 0.47)	(D07, 0.46)
(D04, 0.45)	(D09, 0.37)
(D01, 0.42)	(D10, 0.32)
(D06, 0.24)	(D02, 0.17)
(D07, 0.19)	(D03, 0.07)
.....	.....
(D05, 0.13)	(D01, 0.05)

Fig. 11 Example for query q="Achlorhydria, Hypopotassemia". There are two groups {Achlorhydria, Hypochlorhydria} and {Hypopotassemia, Hypokalemia}. The top-1,2 result is q'="Achlorhydria, Hypokalemia".

9 Experiments

In this section, we report an extensive experimental evaluation of our algorithms, using four real-life data sets. Our experiments were conducted to verify the efficiency and scalability of all our top-k, m algorithms.

Implementation and Environment. All the algorithms were implemented in Java and the experiments were performed on a computer with a 4th generation Intel i7-4770 processor, 16 GB RAM running Ubuntu 14.04.1.

Data sets. We use four datasets including NBA[7], Yahoo! YQL[8], DBLP[9], and PubMed[10] to test the efficacy of top-k,m algorithms in the real world. Table 1 summarizes the characteristics of the four datasets. NBA and Yahoo! YQL datasets were employed to evaluate the top-k, m algorithms *with* and *without* random accesses, while DBLP and PubMed

[5] http://www.ncbi.nlm.nih.gov/pubmed
[6] http://www.nlm.nih.gov/mesh/meshhome.html
[7] http://www.nba.com/
[8] http://developer.yahoo.com/yql/console/
[9] http://dblp.uni-trier.de/xml
[10] http://www.ncbi.nlm.nih.gov/pubmed

Datasets	# of objects	# of groups		group size		# of combinations	
		max	avg	max	avg	max	avg
YQL	100,000	3	3	150	12	3,375,000	1,728
NBA	31,200	5	5	32	6	33,554,432	7,776
DBLP	3,736,406	7	2.6	12	5	371,292	327
PubMed	18,232,012	9	4.3	164	7.9	3,340,631,305	7,241

Table 1 Datasets and their characteristics

datasets were utilized to test the XML top-k, m algorithm and no random accesses top-k, m algorithms respectively.

- **NBA Dataset.** We downloaded the data of 2010–2011 pre-season in NBA for the "*Point Guard*", "*Shooting Guard*", "*Small Forward*", "*Power Forward*" and "*Center*" positions. The original data set contains thirteen dimensions, such as opponent team, shots, assists and score. We normalized the score of the data into $[0, 10]$ by assigning different weights to each dimension. There are five groups, and the average size of each group is about 6.
- **YQL Dataset.** We downloaded data about the hotels, restaurants, and entertainments from Yahoo! YQL[3]. The goal of the top-k,m queries is to recommend the top-k combinations of hotels, restaurants, and entertainments according to users' feedback. There are three groups, and the average size of each group is around 12.
- **DBLP Dataset.** The size of DBLP is about 127M. In order to generate meaningful query candidates, we obtained 724 synonym rules about the abbreviations and full names for computer science conferences and downloaded Babel[11] data including $9,136$ synonym pairs about computer science abbreviations and acronyms. Regarding to the real-world user queries, the most recent $1,000$ queries are selected from the query log of a DBLP online demo, out of which 219 queries (with an average length of 3.92 keywords) are selected to form a pool of queries that need refinement. Finally, we randomly picked 186 queries that have meaningful results to test our algorithms. Here we show 5 sample XML keyword refinements as follows.
  Q_1:{thomason, huang} is refined by substituting "thomas" for "thomason".
  Q_2:{philipos, data, base} can be refined as {philipos, database}.
  Q_3:{XML, key, word, application, 2008} is refined by deleting "2008", followed by a merging of "key" and "word".
  Q_4:{world, wild, web, search, engine, 2007}, which is refined by either adopting "world, wild, web" $\rightarrow$ "www" or deleting "2007".
  Q_5:{object seek} which can be refined to be {object search}.
- **PubMed Dataset.** PubMed has over 18 million citations for biomedical literature from National Library of Medicine premier bibliographic database, life science journals, and also online books. PubMed citation includes the fields of biomedicine and health, covering portions of the life sciences, behavioral sciences, chemical sciences, and bioengineering. In addition, all the citations in PubMed are labeled by different categories, which are descriptors from MeSH (National Library of Medicine's controlled vocabulary thesaurus). MeSH contains about $27,149$ descriptors and over $218,000$ entry terms that assist in finding the most appropriate MeSH Heading, for example, "*Vitamin C*" is an entry term to "*Ascorbic Acid*". We cluster relevant MeSH Headings together as *rules* to build "groups" for keywords. In Table 2, we list some example rules that are collected by domain experts and can be searched in MeSH[12].

	Synonym rules
Teeth-disease related rules	Odontome→Abnormality Teeth
	Toxoplasmosis→Toxoplasma gondii Infection
	Trismus→Lockjaw
	Trismus→Masseter Spasm
Scrofulas related rules	Scrofulas→Tuberculous
	Scrofulas→Cervical Lymphadenitis
	Epiloia→Tuberose Sclerosis
	Cystoliths→Bladder Stone

Table 2 Example synonym rules collected in PubMed.

# lists	5	10	15	20	25	30
# combinations	3125	100000	759375	3200000	9765625	24300000
# pruned combinations	1875	80000	494325	2332800	7604375	19756800
Pruning Ratio	60.0%	80.0%	65.1%	72.9%	77.9%	81.3%

Table 3 The performance of optimization to reduce combinations on NBA dataset

Metrics. Our performance metrics are (1) running time: the cost of the overall time in executing top-k,m queries; (2) access number: the total number of tuples accessed by both sorted access and random access and (3) number of processed combinations: the total number of combinations processed in memory.

We inspected the results returned from all tested algorithms and found that their results are all the same, which verifies the validity of our algorithms. Each experiment was repeated over 10 times and the average numbers are reported here.

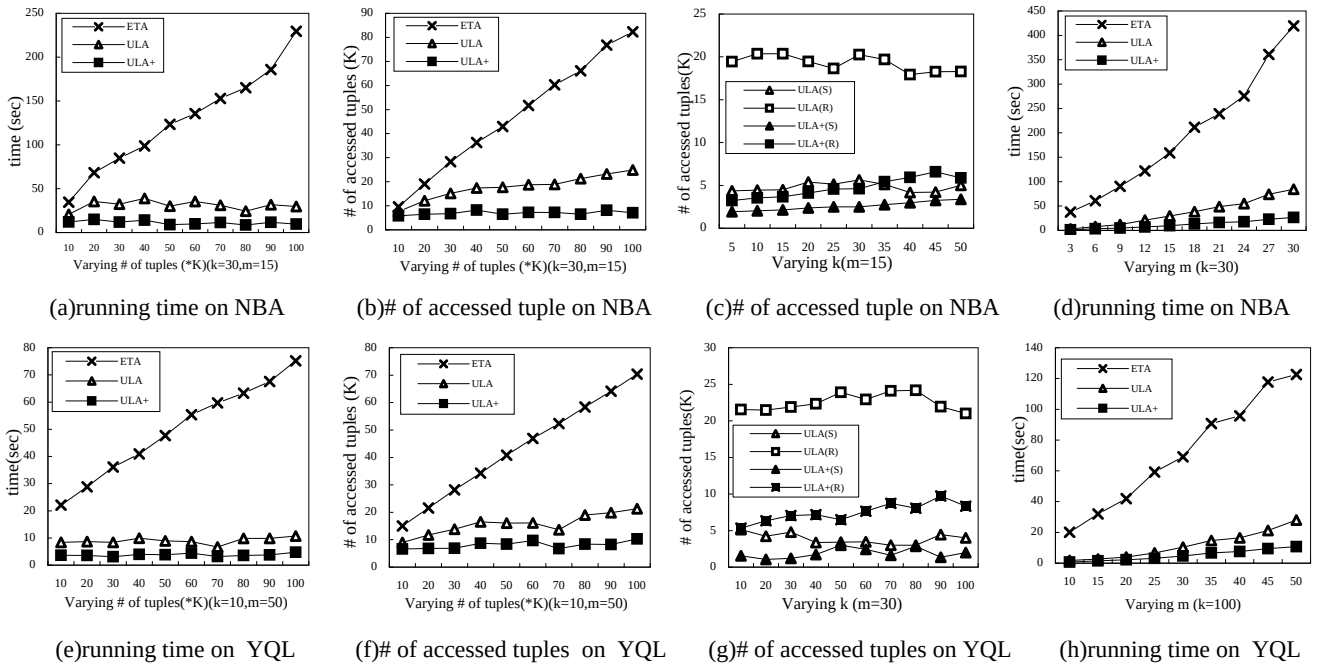
9.1 Experiments of algorithms with random accesses

Here we illustrate the performance of algorithms (ETA, ULA and ULA$^+$) on NBA and YQL dataset by varying parameters k, m, and the data size. In addition, we also deeply study the performance of different optimizations.

Scalability with database size. We evaluated the scalability of our algorithms with varying the number of tuples from 10k

[11] http://www.wonko.info/ipt/babel.htm

[12] http://www.nlm.nih.gov/mesh/meshhome.html



(a)running time on NBA (b)# of accessed tuple on NBA (c)# of accessed tuple on NBA (d)running time on NBA

(e)running time on YQL (f)# of accessed tuples on YQL (g)# of accessed tuples on YQL (h)running time on YQL

Fig. 12 Experiments in NBA and Yahoo! YQL datasets

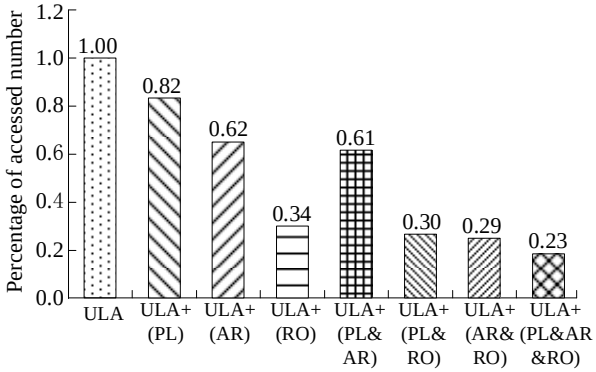
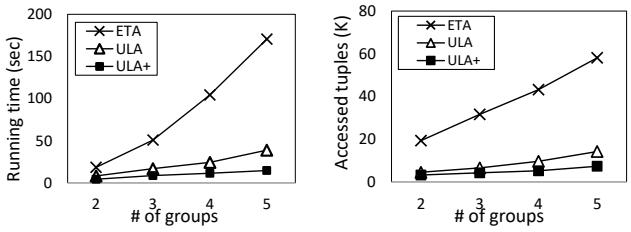


Fig. 13 The performance of different optimizations on YQL.



(a) (b)

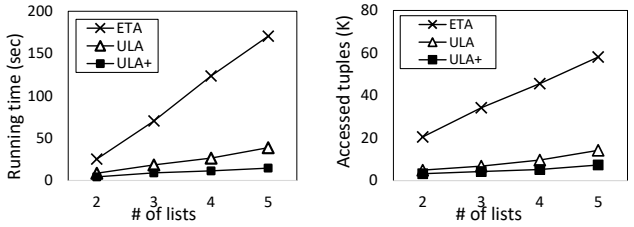
Fig. 14 Effect of the number of groups for top-k,m algorithms with random accesses (NBA).

to 100k in both data sets. As shown in Figure 12(a)(b), both ULA and ULA$^+$ expose an amazingly stable performance without any significant fluctuation both in running time and number of accessed tuples while ETA scales linearly with the size of the database in NBA dataset. And in general, the execution time of ULA$^+$ outperforms ETA by 1-2 orders of magnitude, which verifies the efficiency of the optimizations. In addition, as we can see in Figure 12(e)(f), the results in YQL datasets are similar to that in NBA dataset.

Performance vs. range of k. In Figure 12(c)(g) we tested the performance with different k. We fixed $m = 15$ and varied k from 5 to 50 in NBA dataset. Similarly, we fixed $m = 30$ and varied k from 10 to 100 in YQL dataset. We found that the number of accesses of ETA is the same over all k values because ETA has to obtain the exact top-m match instances for each combination independent of k, while the ULA and ULA$^+$ algorithms significantly outperform the ETA. The reason for this improvement is that ULA and ULA$^+$ can stop earlier without computing the exact top-m match instances for each combination. One interesting observation is that though the YQL dataset has more number of objects than NBA dataset, the running time of top-k,m algorithms in YQL is much smaller than that in NBA (see Figure 12) . This is because YQL has less number of combinations (see Table 1) than that of the NBA, which acts as a key factor to impact the run-time performance.

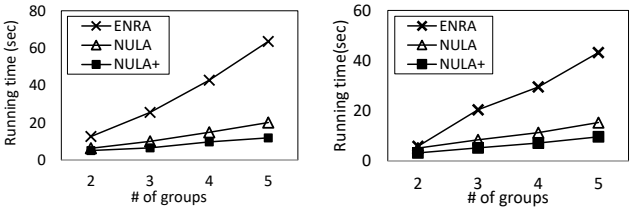
Performance vs. range of m. The results with increasing m from 3 to 30 in NBA data and from 10 to 50 in YQL data are shown in Figure 12(d)(h), respectively. In general, both ULA and ULA$^+$ are 1 to 2 orders of magnitude more efficient than ETA method. In addition, ULA$^+$ is more efficient than ULA, which verifies the effects of our optimizations.

Performance vs. number of groups. We vary the number of groups from 2 to 5 while fixing the number of lists in each group to be 5 in NBA dataset. As shown in Figure 14,



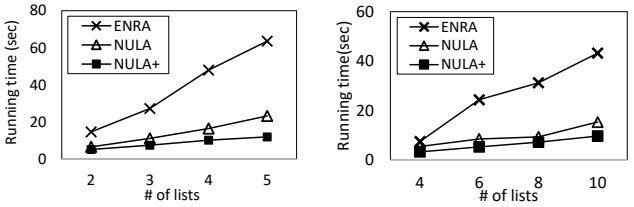
(a) (b)

Fig. 15 Effect of the number of lists for top-k,m algorithms with random accesses (NBA).



(a) NBA (b) PubMed

Fig. 16 Effect of the number of groups for top-k,m algorithms with no random accesses.



(a) NBA (b) PubMed

Fig. 17 Effect of the number of lists for top-k,m algorithms with no random accesses.

both ULA and ULA$^+$ achieve very stable performance and perform better than ETA. Note that ETA scales linearly with the number of groups, as ETA needs to compute the exact score for each combination. Instead ULA and ULA$^+$ only need to compute lower and upper bounds, which makes them less sensitive to the number of groups than ETA.

Performance vs. number of lists. In addition to the evaluation of the impact of the number of groups, here we vary the number of lists from 2 to 5 while keeping the number of groups to be 5 in NBA dataset in order to study the effect of the number of lists. Figure 15 shows that ULA and ULA$^+$ again have better performance than ETA and behave very stably, while ETA scales linearly with the number of lists. The key reason is that ETA needs to compute the exact score for each of the combination, while ULA and ULA$^+$ only calculate the upper and lower bounds of each combination.

Effect of the optimizations in ULA$^+$. We performed experiments to investigate the effects of four different optimizations in ULA$^+$. We fixed the parameters $k = 10$, $m = 30$ and the number of tuples is 100k. First, to evaluate the approach of pruning useless combinations introduced in Section 5.3, we plotted Table 3 to show that the number of combinations processed in memory by our optimized algorithm is far less than that of ULA on NBA dataset. More than 60% combinations are pruned without computing their bounds on NBA dataset, thus significantly reducing the computational and memory costs.

To evaluate the effects of Lemma 4 to 6, Figure 13 is plotted to evaluate the performance of different optimizations in terms of the number of accessed tuples. In particular, ULA$^+$(PL) uses Lemma 4 to prune the whole lists to avoid useless access; ULA$^+$(AR) applies Lemma 5 to avoid random access in some lists; and ULA$^+$(RO) employs Lemma 6 to prevent random access for some tuples. In Figure 13(a), the first three pairs of bars show the results to measure three optimizations individually, while the others are actually a combination of multiple optimizations. As shown, the combination of all optimizations has the most powerful pruning capability, reducing the accesses for almost 80%. It is similar in NULA$^+$, the combination of all optimizations (Lemma 4 and Lemma 8) achieves the most powerful pruning power by reducing the accesses for around 21%. The overall pruning power of ULA$^+$ is higher than that of NULA$^+$. This is because ULA$^+$ can use two more optimizations relying on random accesses, i.e., AR (Lemma 5) and RO (Lemma 6) optimizations.

XML keyword refinement We ran the experiments to test the scalability and efficiency of XETA, XULA and XULA$^+$ algorithms on DBLP dataset. In Figure 12(i)(j), we varied the size of DBLP dataset from 20% to 100% while keeping k=3, m=2. As expected, both XULA and XULA$^+$ perform better than XETA and scale well in both running time and number of accesses. In Figure 12(k)(l), we varied k from 1 to 5 while fixing $m = 2$ and 100% data size. As shown, both XULA and XULA$^+$ are far more efficient than XETA, and XULA$^+$ accesses 74.2% fewer objects than XULA and saves more than 35.1% running time, which indicates the effects of our optimizations.

9.2 Experiments of algorithms with *no* random accesses

Here we illustrate the performance of algorithms (ENRA, NULA and NULA$^+$) on NBA, YQL and PubMed datasets by varying parameters k, m, and the data size.

Scalability with database size. We evaluated the scalability of our algorithms by varying the number of tuples from 10k to 100k in both data sets. As shown in Figure 18(a,b,e,f,i,j), NULA and NULA$^+$ scale better than ENRA both in terms of both running time and the number of accessed tuples, since NULA and NULA$^+$ can stop earlier than ENRA, which needs to compute the exact score for each combination. In addition, the performance of NULA$^+$ is better than that of NULA in our experiments, since NULA$^+$ adopts one opti-

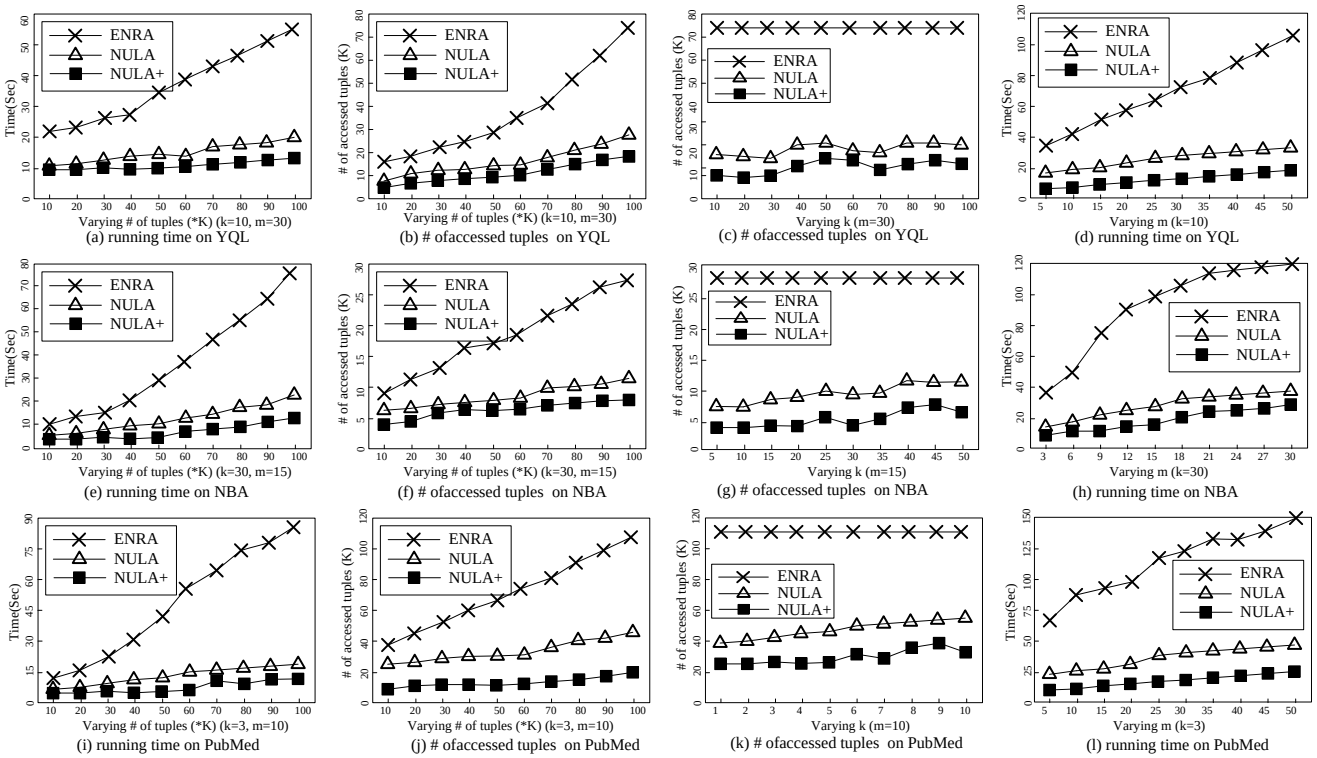


Fig. 18 Experimental results of top-k,m algorithms with *no* random accesses in NBA, Yahoo! YQL and PubMed datasets

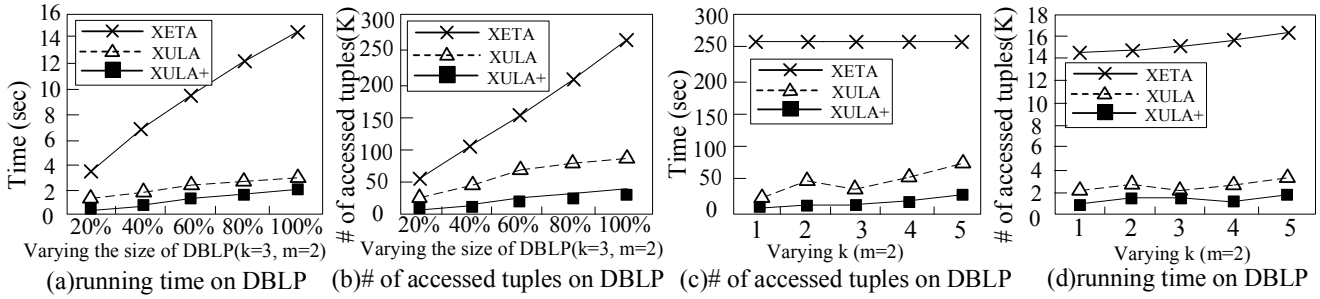


Fig. 19 Experiments on DBLP datasets

mization method (Lemma 8) to reduce the number of the sorted accesses. Note that, the number of accessed tuples is large in PubMed dataset, because over 50% citations do not cover all the input keywords, i.e., it is harder to obtain full score for each match instance.

Performance vs. range of k. In Figure 18(c)(g)(k) we tested the performance of algorithms by (i) varying k from 10 to 100 when $m = 30$ in YQL dataset; (ii) varying k from 5 to 50 while fixing $m = 15$ in NBA dataset; and (iii) varying k from 1 to 10 when $m = 10$ in PubMed dataset. As shown, NULA and NULA$^+$ algorithms significantly outperform the ENRA. In addition, we observe that the number of accesses of ENRA remains the same over all k values, while that of NULA and NULA$^+$ fluctuate over different k. The main reason is that: (i) ENRA has to obtain the exact top-m match instances for each combination independent of k; and (ii) NULA and NULA$^+$ find the k combinations whose lower bounds are larger than the others' upper bounds, which means that the top-k,m query possibly requires accessing more tuples than the top-k', m (k'>k) query does. For example, see the points $k = 50$, $k = 60$, and $k = 70$ in Figure 18(c).

Performance vs. range of m. The results with increasing m from 10 to 50 in YQL and PubMed datasets, from 3 to 30 in NBA dataset are shown in Figure 18(d)(h)(l), respectively. In general, both NULA and NULA$^+$ are much more efficient than ENRA method. In addition, NULA$^+$ is more efficient than NULA, which verifies the effects of our optimizations.

Performance vs. number of groups. To test the effect of the number of groups for top-k,m algorithms with no random accesses, we vary the number of groups from 2 to 5 in NBA (by fixing the number of lists with 5 in each group) and PubMed (by fixing the number of lists with 10

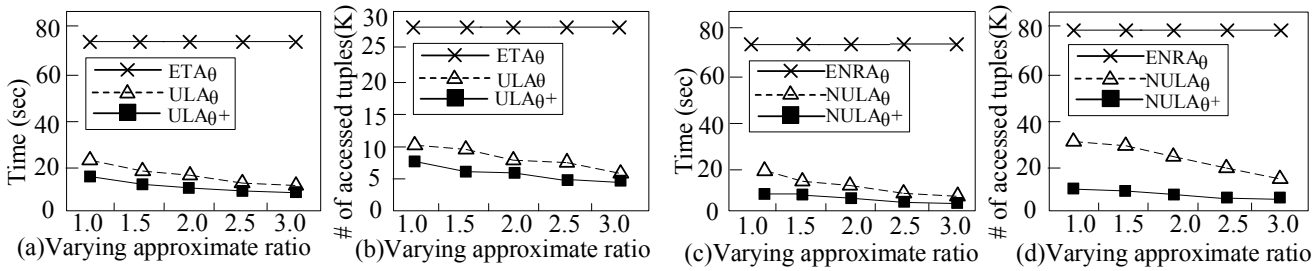


Fig. 20 Experiments of approximate algorithms

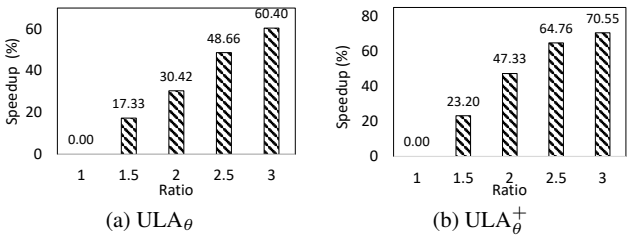


Fig. 21 The speedup of different ratios

θ	1.0	1.5	2.0	2.5	3.0
NBA (p)	100%	97%	94%	86%	81%
YQL (p)	100%	98%	91%	88%	73%

Table 4 Precision under different ratios

in each group) datasets. Figure 16 shows that both NULA and NULA$^+$ achieve stable performance and perform better than ENRA. Since ENRA needs to compute the exact top-m match instances for each combination, ENRA scales linearly with the number of groups.

Performance vs. number of lists. To study the effect of the number of lists for the top-k,m algorithms with no random accesses, we vary the number of lists from 2 to 5 in NBA and 4 to 10 in Pubmed datasets. As shown in Figure 17, ENRA scales linearly with the number of lists while both NULA and NULA$^+$ achieve stable performance and perform higher than ENRA.

9.3 Experiments of approximate algorithms

Finally, we evaluate the performance of all the approximate top-k,m algorithms. Figure 20(a) and (b) show the running time and the number of accesses on NBA dataset for ETA$_\theta$, ULA$_\theta$ and ULA$_\theta^+$ with varying approximate ratios (from 1.0 to 3.0). As shown, ULA$_\theta^+$ performs the best, which outperforms ETA$_\theta$ by around 5 times. In addition, with larger approximate ratios, ULA$_\theta$ (resp. ULA$_\theta^+$) accesses fewer tuples and has higher performance. However, ETA$_\theta$ remains the same for different approximate ratios, since it always need to get all the top-m objects. Figure 20(c) and (d) demonstrate the performance of ENRA$_\theta$, NULA$_\theta$ and NULA$_\theta^+$ on PubMed dataset with different approximate ratios. The results are similar to those in Figure 20(a) and (b).

Improvement vs. approximate ratios. To evaluate the performance gain from different approximate ratio settings, we plotted Figure 21 to show how much speed up obtained (in percentage) with different approximate ratios. The speed up percentage ρ is defined as $\rho = \frac{t-t_\theta}{t}$, where t is the running time of a top-k,m algorithm and t_θ is that of its approximate version. As shown in Figure 21, the higher speed up is obtained with larger approximate ratios. For example, ULA achieves 30.43% speed up with $\theta = 2.0$ and 60.40% improvement with $\theta = 3.0$. We also observed that the performance improvement tends to be stable at some points, which means continuously increasing approximate ratios may not keep gaining performance benefits.

Accuracy under different ratios. To study the result quality of the approximate top-k,m algorithms, Table 4 shows the precision of our top-k,m algorithms in NBA and YQL datasets (k=100 and m=10). As shown, our algorithms achieve high result quality. For example, in NBA dataset, even with an approximate ratio 3, the precision is still above 80%.

Summary. From the experimental results and performance evaluation we found the following: (1) Our algorithms (with and without random accesses) are scalable and robust with the number of tuples, data size, m and k, the number of groups and the number of lists. (2) The optimizations in ULA$^+$ and NULA$^+$ speed up the query processing significantly. (3) Our algorithms outperform baselines in both accurate and approximate environment.

10 Conclusion and future work

In this article, we proposed a new problem called top-k,m query evaluation. We developed a family of efficient top-k,m algorithms ULA, ULA$^+$, NULA and NULA$^+$ with and without random accesses. We provided the corresponding approximate version for each of them. We theoretically proved the optimality of each algorithm. To demonstrate the applicability of the top-k,m problems, we applied our algorithms to the query refinement problem in a biomedical database. Finally, we conducted comprehensive experiments on four real-life datasets to verify the efficiency of our algorithms.

This is an initial investigation on top-k,m problems, and we plan to extend this work in several directions. One of them is to apply top-k,m algorithms on probabilistic databases. We are also interested in studying the impact of modern hardwares, e.g., SSD, on top-k,m problems when lists are stored in secondary storage. For example, the fact that SSD supporting high speed random accesses may allow algorithms to perform on-disk skip.

References

1. B. Babcock, M. Datar, and R. Motwani. Sampling from a moving window over streaming data. In *SODA*, pages 633–634, 2002.
2. H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum. Io-top-k: Index-access optimized top-k query processing. In *VLDB*, pages 475–486, 2006.
3. N. Bruno, S. Chaudhuri, and L. Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *ACM TODS*, 27(2):153–187, 2002.
4. N. Bruno, L. Gravano, and A. Marian. Evaluating top-k queries over web-accessible databases. In *ICDE*, pages 369–380, 2002.
5. K. C.-C. Chang and S.-w. Hwang. Minimal probing: supporting expensive predicates for top-k queries. In *SIGMOD*, pages 346–357, 2002.
6. L. J. Chen and Y. Papakonstantinou. Supporting top-k keyword search in xml databases. In *ICDE*, pages 689–700, 2010.
7. M. Dylla, I. Miliaraki, and M. Theobald. Top-k query processing in probabilistic databases with non-materialized views. In *ICDE*, pages 122–133, 2013.
8. R. Fagin. Combining fuzzy information from multiple systems. In *PODS*, pages 216–226, 1996.
9. R. Fagin, R. Kumar, and D. Sivakumar. Comparing top k lists. *SIDMA*, 17(1):134–160, 2003.
10. R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *JCSS*, 66(4):614–656, 2003.
11. W. Fan, X. Wang, and Y. Wu. Diversified top-k graph pattern matching. *PVLDB*, 6(13):1510–1521, 2013.
12. J. Feng, G. Li, and J. Wang. Finding top-k answers in keyword search over relational databases using tuple units. *TKDE*, 23(12):1781–1794, 2011.
13. J. Guntzer, W.-T. Balke, and W. Kießling. Towards efficient multi-feature queries in heterogeneous environments. In *ITCC*, pages 622–628, 2001.
14. U. Güntzer, W. Balke, and W. Kießling. Optimizing multi-feature queries for image databases. In *VLDB*, pages 419–428, 2000.
15. R. He, C. Lin, and J. McAuley. Fashionista: A fashion-aware graphical system for exploring visually similar items. In *WWW*, pages 199–202, 2016.
16. R. He, C. Lin, J. Wang, and J. McAuley. Sherlock: Sparse hierarchical embeddings for visually-aware one-class collaborative filtering. In *IJCAI*, pages 3740–3746, 2016.
17. M. Hua, J. Pei, A. W. Fu, X. Lin, and H. Leung. Top-k typicality queries and efficient query answering methods on large databases. *VLDB J.*, 18(3):809–835, 2009.
18. I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Joining ranked inputs in practice. In *VLDB*, pages 950–961, 2002.
19. I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. In *VLDB*, pages 754–765, 2003.
20. I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. *VLDB Journal*, 13(3):207–221, 2004.
21. I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *CSUR*, 40(4):11, 2008.
22. C. Li, K. Chen-Chuan Chang, and I. F. Ilyas. Supporting ad-hoc ranking aggregates. In *SIGMOD*, pages 61–72, 2006.
23. J. Li, C. Liu, R. Zhou, and W. Wang. Top-k keyword search over probabilistic xml data. In *ICDE*, pages 673–684, 2011.
24. X. Lian and L. Chen. Shooting top-k stars in uncertain databases. *VLDB J.*, 20(6):819–840, 2011.
25. E. H.-C. Lu, C.-Y. Chen, and V. S. Tseng. Personalized trip recommendation with multiple constraints by mining user check-in behaviors. In *SIGSPATIAL GIS*, pages 209–218, 2012.
26. J. Lu, P. Senellart, C. Lin, X. Du, S. Wang, and X. Chen. Optimal top-k generation of attribute combinations based on ranked lists. In *SIGMOD*, pages 409–420, 2012.
27. N. Mamoulis, M. L. Yiu, K. H. Cheng, and D. W. Cheung. Efficient top-k aggregation of ranked inputs. *TODS*, 32(3):19, 2007.
28. A. Marian, S. Amer-Yahia, N. Koudas, and D. Srivastava. Adaptive processing of top-k queries in xml. In *ICDE*, pages 162–173, 2005.
29. S. Michel, P. Triantafillou, and G. Weikum. Klee: A framework for distributed top-k query algorithms. In *VLDB*, pages 637–648, 2005.
30. A. Natsev, Y.-C. Chang, J. R. Smith, C.-S. Li, and J. S. Vitter. Supporting incremental join queries on ranked inputs. In *VLDB*, volume 1, pages 281–290, 2001.
31. S. Nepal and M. V. Ramakrishna. Query processing issues in image (multimedia) databases. In *ICDE*, pages 22–29, 1999.
32. M. Qiao, L. Qin, H. Cheng, J. X. Yu, and W. Tian. Top-k nearest keyword search on large graphs. *PVLDB*, 6(10):901–912, 2013.
33. S. Ranu, M. X. Hoang, and A. K. Singh. Answering top-k representative queries on graph databases. In *SIGMOD*, pages 1163–1174, 2014.
34. C. Re, N. Dalvi, and D. Suciu. Efficient top-k query evaluation on probabilistic data. In *ICDE*, pages 886–895, 2007.
35. J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *JACM*, 27(4):701–717, 1980.
36. S. Shang, R. Ding, B. Yuan, K. Xie, K. Zheng, and P. Kalnis. User oriented trajectory search for trip recommendation. In *EDBT*, pages 156–167, 2012.
37. M. A. Soliman, I. F. Ilyas, and K. C.-C. Chang. Top-k query processing in uncertain databases. In *ICDE*, pages 896–905, 2007.
38. M. A. Soliman, I. F. Ilyas, and K. C.-C. Chang. Probabilistic top-k and ranking-aggregate queries. *TODS*, 33(3):13, 2008.
39. M. Theobald, R. Schenkel, and G. Weikum. An efficient and versatile query engine for topx search. In *VLDB*, pages 625–636, 2005.
40. M. Theobald, G. Weikum, and R. Schenkel. Top-k query evaluation with probabilistic guarantees. In *VLDB*, pages 648–659, 2004.
41. R. Varadarajan, F. Farfán, and V. Hristidis. Comparing top-k XML lists. *Inf. Syst.*, 38(6):820–834, 2013.
42. S. Yang, F. Han, Y. Wu, and X. Yan. Fast top-k search in knowledge graphs. In *ICDE*, pages 990–1001, 2016.
43. Z. Yang, A. W. Fu, and R. Liu. Diversified top-k subgraph querying in a large graph. In *SIGMOD*, pages 1167–1182, 2016.
44. M. L. Yiu, N. Mamoulis, and V. Hristidis. Extracting k most important groups from data efficiently. *DKE*, 66(2):289–310, 2008.
45. X. Zhang and J. Chomicki. Semantics and evaluation of top-k queries in probabilistic databases. *Distributed and parallel databases*, 26(1):67–126, 2009.
46. R. Zhu, Z. Zou, and J. Li. Towards efficient top-k reliability search on uncertain graphs. *KAIS*, 50(3):723–750, 2017.